FINAL

2

IDA MEMORANDUM REPORT M-513

TOWARDS SDS TESTING AND EVALUATION: A COLLECTION OF RELEVANT TOPICS

Compiled and Edited by
Bill R. Brykczynski
Christine Youngblut

July 1989

Prepared for
Strategic Defense Initiative Organization

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 1989 July	**3. REPORT TYPE AND DATES COVERED** Final

4. TITLE AND SUBTITLE Towards SDS Testing and Evaluation: A Collection of Relevant Topics (U)	**5. FUNDING NUMBERS** MDA 903 89 C 0003 T-R2-597.21
6. AUTHOR(S) Bill R. Brykczynski, Christine Youngblut	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses 1801 N. Beauregard St. Alexandria, VA 22311-1772	**8. PERFORMING ORGANIZATION REPORT NUMBER** IDA Memorandum Report M-513
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) SDIO T&E Room 1E149, The Pentagon Washington, D.C. 20301-7100	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution.	**12b. DISTRIBUTION CODE** 2A

13. ABSTRACT *(Maximum 200 words)*

IDA Memorandum Report M-513 is a compendium of position statements received at a two-day invitational workshop hosted by the Institute for Defense Analyses (IDA) in September 1988. The purpose of the workshop was (1) to determine the critical gaps in software testing and evaluation technology and (2) to assess current research efforts toward resolving these deficiencies. The specific focus of the workshop was technology for the testing and evaluation of the Strategic Defense System (SDS) software. The workshop was designed to bring together leading researchers in the field of software testing and evaluation and government agencies involved in major software development efforts. The participants were divided into four groups, respectively addressing the areas of testing and analysis, formal verification, measurement, and reliability assessment.

14. SUBJECT TERMS Software Testing and Evalution; Strategic Defense Software (SDS); Software Development; Testing and Analysis; Formal Verification; Reliability.	**15. NUMBER OF PAGES** 160
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

IDA MEMORANDUM REPORT M-513

TOWARDS SDS TESTING AND EVALUATION: A COLLECTION OF RELEVANT TOPICS

Compiled and Edited by
Bill R. Brykczynski
Christine Youngblut

July 1989

IDA
INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 89 C 0031
Task T-R2-597.21

TABLE OF CONTENTS

PREFACE

In September 1988 the Institute for Defense Analyses (IDA) held a two-day, invitational workshop to determine the critical gaps in software testing and evaluation technology and to assess current research efforts towards resolving these deficiencies. The specific focus of the workshop was technology for the testing and evaluation of Strategic Defense System (SDS) software. While the primary intent was to ensure a sufficient technology base for cost-effective testing and evaluation of full-scale development SDS software (expected to begin in 1990's), it is recognized that software testing and evaluation is a major concern for all DOD computer-based systems.

The workshop was designed to bring together leading researchers in the field of software testing and evaluation, and government agencies involved in major software development efforts. The participants were divided into four groups, respectively addressing the areas of testing and analysis, formal verification, measurement, and reliability assessment. In order to ensure that maximum advantage was taken of the available time, those participants actively engaged in research had been asked to provide position statements prior to the workshop. Specifically, these researchers were asked to describe:

- Their views of critical gaps in technology, and
- The status of their current research efforts.

IDA Memorandum M-513, *Towards SDS Testing and Evaluation: A Collection of Relevant Topics* is a compendium of position statements received.

An extensive bibliography of software testing and evaluation material is given in IDA Memorandum M-496. Another related document is IDA Paper P-2132, *SDS Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks*. These three documents, Paper P-2132, Memorandum M-496, and Memorandum M-513, were prepared for the Strategic Defense Initiative Organization.

Opening Session

LTC J. Price

Panel: Software Testing and Analysis

J. Salasin, Chair

Panelists:

L.A. Clarke
L.K. Dillon
W.E. Howden
D.C. Luckham
L.J. Osterweil
D.J. Richardson
R.N. Taylor
L.E. White
S.J. Zeil

Software Validation: Directions for the Future

Lori A. Clarke
Software Development Laboratory
University of Massachusetts
Amherst, Massachusetts 01003

The software validation area has matured considerably over the years. The optimism of the seventies has given way to a more realistic outlook on the enormous scope and complexity of the problem of producing reliable software. Instead of arguing over whether verification or validation is best, we are concerned with how these two approaches can complement each other. In fact there appears to be general agreement that a validation system powerful enough to provide the high degree of reliability assurance being sought will require extensive integration of a number of diverse analysis techniques.

Although there has been considerable work, and progress, on analysis techniques for sequential programs, it is incorrect to assume that what remains to be done to automate these techniques is primarily an engineering problem. Automating a well chosen and integrated set of analysis capabilities into a coherent and powerful testing system is an enormously complex task, raising many challenging research issues. In fact, it is just this task that has been the major thrust of much of our recent research in the Software Development Laboratory. In particular, to determine just which techniques should be included in a testing system we have evaluated a number of different techniques [Clar85,Rich88]. In addition, the Arcadia [Tayl88] and TEAM [Clar88] projects have been investigating strategies and support mechanisms for automating integrated collections of analysis techniques. Based on the insights gained from this work, this report outlines some of the major challenges that need to be addressed in automating a powerful, integrated collection of validation techniques. These include:

- technique integration,
- generic tool design,
- incremental support,
- language independence,
- user interaction models, and
- environment support.

The problems and payoffs from each of these is briefly described.

Support for Automation: Although there has been substantial work and progress on sequential techniques, many important questions remain unanswered. Most importantly, which of the sequential techniques should be employed and how should they be applied to be both efficient and powerful? These are important questions that must be addressed before selecting techniques and integrating them into a validation system. One of the lessons we have learned from earlier work is that loose integration, where one technique sequentially follows another, will usually not lead to the increased fault detection that is being sought. Instead, different testing techniques, to be effective, must be tightly integrated. That is, intermediate results from one technique must be made available to another technique, which might then, after building upon this information, return some intermediate information about itself, and so on. Examples of techniques that are exploring tight integration are the RELAY model [Rich88], the Mothra system [DeMi88], and Young's and Taylor's [Youn88b] concurrency system. Without such tight integration, a testing system will not only be extremely inefficient but will fail to provide the rigorous fault detection capabilities that are desired. Considerable more research is required to understand the ways in which testing techniques should be combined. Empirical evaluation is one way in which such insight can be gained. Many analysis techniques, however, can be compared analytically, providing unbiased evaluation results. Both empirical and analytic evaluation provide valuable information about effectiveness and should be pursued in trying to determine the most effective combinations of techniques.

Generic tool design: In trying to understand how techniques can be integrated, we have been frustrated by how inflexible many of the automated tools are. This is not surprising since these tools are usually prototypes intended to explore the feasibility of an approach. On the other hand, experimentation with a tool often leads to the desire to modify it. Usually these prototype tools are too rigidly constructed to support the desired experimentation. Moreover, in an automated validation system, one would expect that new techniques would be added and new uses for old techniques would evolve. One approach to supporting such flexibility is to make each tool as generic as possible. In the TEAM environment, for example, we are finding that the careful design of generic capabilities leads to extensive reuse, often in ways that were unforeseen by the developers. Unfortunately, there is considerable up-front cost in determining the dimensions of flexibility of a technique and designing a generic tool. A validation system will be of limited use, however, if this type of effort is not expended in its tool design and development.

One area that is particularly in need of more flexible support is reasoning facilities. Validation tools need a number of different approaches to reasoning about generated information. These include simplification, inequality solving, and theorem proving. Most of the reasoning systems come with a predisposed set of assumptions that unfortunately limit their applicability in software analysis. For example, simplifiers usually support the general rules of arithmetic, but in software development there may be additional rules depending on the application domain. Reasoning tools that allow rules and heuristics to be specified are needed to support validation.

Incremental analysis: Also related to the issue of flexibility is the need for techniques to be designed to be applied incrementally to possibly incomplete representations. Most tools currently work on complete systems, programs, or modules. Systems, however, start out incomplete and are developed incrementally. It is unfortunate that developers must wait until they have completed a whole "unit" before they can get feedback from available analysis tools. To alleviate this problem, analysis techniques should be extended to deal with incomplete information. This implies recognizing not only consistent and inconsistent states, but a spectrum of "potentially" (in)consistent states as well. In terms of incremental support, it is desirable to be able to efficiently reassess a system after any change. The impact of a change can vary widely, however, from pervasive to self-contained. If a change has a small impact, it is often inefficient to re-analyze the unaffected portions of the systems. For large systems or for costly analysis techniques, the cost of doing monolithic, instead of incremental, analysis might be so prohibitive that analysis is done infrequently or circumvented altogether. Thus, providing analysis for large systems, whenever it is requested, requires that tools be developed to respond well to incomplete systems and to re-analyze only the effected portions of a system. As similar goals in language processing have shown, developing approaches that can incrementally handle incomplete representations is quite challenging.

Language independence: Tools are usually constructed to work on a particular programming language. Software development efforts, however, involve several languages, such as functional and detailed design languages and perhaps a few programming languages. Today different analysis techniques are applied to each type of notation, leading to a hodge-podge of approaches. With the appropriate underlying model, most analysis techniques can be designed to be applicable to a number of different languages, even languages at very different levels of abstraction. While determining this underlying model and building tools to exploit this generality adds overhead to tool development, the gain is a uniform set of analysis capabilities applicable to many of the different phases of the lifecycle. This is an important step toward achieving pre-implementation analysis support.

User interaction models: There have been many recent advances in the area of user interface management systems that can be exploited by analysis tools. These systems provide capabilities to ease the burden of implementing a particular user interaction model. The problem with analysis tools to date is that they provide user-unfriendly interaction models. For the most part, the interaction models tend to be technique-oriented instead of user-oriented, requiring users to be very knowledgeable about the techniques being applied. This is unfortunate since many of the techniques are non-intuitive. Another problem is that many techniques inundate the user with information. Thus, methods are needed for filtering information to the user and allowing the user to input and receive (filtered) information in a more natural form. The problem of developing acceptable user interaction models is further complicated when one considers the diversity of the tools that should be provided in a validation system. Approaches for providing interface uniformity across tools (e.g., [Youn88a]) need to be explored further.

Environment support: Related to the issue of an interaction model is the capability for describing analysis processes. Instead of requiring the user to know about each and every analysis tool of interest and to know the acceptable ways in which these tools can be used, process descriptions or programs [Oste87] define these interactions. It is then the job of the environment to support and enforce these prescriptions. As tools are added to the environment, new processes will need to be described and old processes will need to be modified to incorporate the new tool capabilities. Support for this kind of flexibility is needed if the validation system is to continue to be viable as new techniques are discovered.

Clearly, much can be done over the next five years to improve significantly the reliability of software. Most of this gain will come about through the careful automation and integration of many of the existing testing and analysis techniques. If done with appropriate concern for flexibility, the resulting validation system will be able to grow and change as new techniques are discovered and as we gain experience with automated versions of existing techniques.

The current state of research in software validation, however, is a far cry from what is needed to provide the high level of reliability assurance required for the SDS. In fact, our science is so premature that we do not currently know how to assess that reliability or even what units of measure to use. Considerably more research is needed in the software validation area to address these issues. This research should be directed by a long term, aggressive research program. Although the benefits from such a program may come too late to address the needs of SDS software currently being developed, it will have a significant impact on the reliability of future software endeavors.

REFERENCES

[Clar85] L.A. Clarke, A. Podgurski, D.J. Richardson, and S.J. Zeil, A Comparison of Data Flow Path Selection Criteria, Eighth International Conference on Software Engineering, London, England, August 1985, pp. 244-251.

[Clar88] L.A. Clarke, D.J. Richardson, and S.J. Zeil, TEAM: A Support Environment for Testing, Evaluation, and Analysis, University of Massachusetts COINS Technical Report 88-41, to appear in the 3rd Conference on Software Development Environments, Cambridge, Massachusetts, November 1988.

[DeMi88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt, An Extended Overview of the Mothra Software Testing Environment, Proceedings of the Second Workshop on Software Testing, Analysis, and Verification, July 1988.

[Oste87] L. Osterweil, Software Processes Are Software Too, Proceedings of the Ninth International Conference on Software Engineering, pp. 2-13, Monterey, CA, March 1987.

[Rich88] D.J. Richardson and M.C. Thompson, The Relay Model of Error Detection and Its Application, Proceedings of the Second Workshop on Software Testing, Analysis, and Verification, July 1988.

[Tayl88] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young, Foundations for the Arcadia Environment Architecture, University of Massachusetts COINS Technical Report 88-43, to appear in the 3rd Conference on Software Development Environments, Cambridge, Massachusetts, November 1988.

[Youn88a] M. Young, R.N. Taylor, D.B. Troup, and C.D. Kelly, Design Principles Behind Chiron: A UIMS for Software Environments, Proceedings of the Tenth International Conference on Software Engineering, pp. 367-376, IEEE, Singapore, April 1988.

[Youn88b] M. Young and R.N. Taylor, Combining Static Concurrency Analysis and Symbolic Execution, to appear in IEEE Transactions on Software Engineering, October 1988.

Critical Gaps in Software Validation

A Position Paper

Laura K. Dillon
University of California
Santa Barbara CA 93106

1. Introduction

There can be little, if any, margin for error in a system that can have catastrophic societal and environmental consequences, such as the proposed Strategic Defense System (SDS). The high degree of reliability required for SDS software, however, cannot be attained using existing software validation and verification technologies. Two fundamental characteristics of SDS software account for its intractability: the massive parallelism inherent in the system and its stringent real-time requirements. While the last decade has brought significant advances in the validation and verification of sequential software systems, very little progress has been made in addressing issues raised by concurrent and real-time systems. Critical issues in the validation of concurrent and real-time systems are discussed below. The discussion focuses on validation, rather than verification, which is being addressed by members of another panel. Suggestions are then made for how technological gaps could be best addressed in the near-term and in the long-term.

2. Validation of concurrent real-time systems

The complexity of writing software increases with the introduction of parallelism and is further complicated with the introduction of real-time constraints. Similarly, the difficulty of validating software increases from sequential to parallel to real-time systems. The large number of potential interactions between asynchronous components of a concurrent system make it hard to establish critical correctness criteria. Real-time systems have the additional problem that they must meet critical performance deadlines. Validation of real-time systems involves demonstrating that the system meets the performance deadlines in every case and, in particular, in the worst case. The problems that general validation techniques have in dealing with concurrency and real-time constraints are discussed briefly below.

Testing

It has long been recognized that testing cannot provide a complete picture of the full range of behaviors that a system (concurrent or sequential) is capable of producing; it reveals only a single possible behavior for each of a finite number of trials. Many system properties, both desirable and undesirable, may never be revealed through test runs. This is especially problematic for real-time systems, where it is necessary to guarantee that critical performance deadlines are met by every possible execution.

While testing is still the primary means for validating sequential software, appropriate testing strategies for dealing with concurrency have yet to be identified. Traditional notions for assessing the adequacy of test data (e.g., path or branch coverage [How75],[Stu77], distinguishability of mutants [SDL78], blindness measures [ZW81], etc.) do not take into account the relative timing of event occurrences involving different processes. The need to test different orders of event occurrences is especially critical in embedded, real-time systems, where tests are done on a host separate from the target. The likelihood of subtle differences in the timing characteristics of the host and target environments make it necessary to test different orders of execution.

The testing of concurrent and real-time software systems is further complicated by the unpredictable nature of their operating environments. It is not unusual for various factors affecting a program's behavior, such as relative processor speeds and communication delays, to vary in an unpredictable and uncontrollable manner. This makes it impossible to faithfully reproduce test runs. Reproducibility is important if the cause of an error is to be identified and the error eliminated.

Moreover, full system testing of SDS software is infeasible due to high costs, and social and environmental implications. While unit testing is possible (and necessary), it cannot substitute for full system testing. Unit testing does not address a fundamental issue: how to determine that system components interact as expected. The complexity of the possible interactions of system components is precisely what makes it so difficult to reason reliably about a concurrent system's properties. This issue accounts for the additional complexity of writing concurrent software, and cannot be overlooked.

Analysis of formal models

The modeling of a concurrent system using an appropriate mathematical formalism, such as Petri nets [Pet77], CCS, [Mil80] or constrained expressions [ADW86], [DAW88], is one approach to validation that can be useful in early phases of software development. Such models provide a basis for analysis of important system properties. For example, analysis of a system model may show that the system is free from deadlock for all possible orders of event occurrences. Petri nets have been extended (e.g., timed Petri nets [MZGT85]) to model real-time systems. An extension of constrained expressions to real-time systems is currently under investigation. However, these and other models of real-time systems are still very experimental.

A mathematical model is designed to highlight the essential features of a system and suppress irrelevant detail. This is necessary to make the analysis tractable. Simplifying assumptions, however, often compromise the precision of the analysis. For instance, static analysis of the synchronization structure of a concurrent system, as described in [Tay83], may report many "behaviors" that cannot occur. The developer must determine which of a (typically large) number of reported behaviors are spurious and which correspond to actual system behaviors. The usefulness of a formal modeling scheme, furthermore, is limited by the level of detail that it can accommodate. This is why most formal models are used only in the early stages of development.

Despite the use of abstraction, the analysis of a formal model of a concurrent system typically involves lengthy chains of inferences, and hence is error-prone. Of course, it may be possible to alleviate this problem using appropriate software development tools. For the most part, however, analysis techniques for concurrent software are in early stages of development, and little automated assistance is currently available. Experience with most techniques for analyzing concurrent software is therefore limited to small-scale "toy" systems.

Another problem with formal analysis is that mathematical formalisms are poorly suited for use by software developers, who do not think in terms of highly abstract mathematical models when designing or implementing software systems. Software developers can be expected to have trouble using unfamiliar mathematical formalisms to describe crucial features of a system under development.

3. Suggested research goals

While significant progress has already been made in the validation and verification of sequential software, there is still much to be done. The next decade will most certainly continue to bring important advancements in the testing, analysis and formal verification of sequential systems. Naturally, validation techniques for concurrent and real-time software will build on validation techniques for sequential software. This fundamental research is therefore of critical importance. However, as indicated above, the most glaring technological gaps are in the validation and formal verification of concurrent real-time systems. The discussion of research goals, therefore, focuses on how to best address the issues raised by concurrency and by real-time constraints.

The inherent limitations of testing concurrent real-time systems, suggests that the major advancements in validation and verification of concurrent real-time software will be in analysis and formal verification, rather than testing. This is especially true of systems, such as the proposed SDS, for which system testing is economically, socially and environmentally unacceptable. Given the state-of-the-art of formal verification and the difficulty of automating formal proofs of correctness, major advancements in the verification of concurrent real-time systems are more likely to come in the long-term (10 to 20 years) than in the near-term (2 to 3 years). Advancements in the analysis of concurrent software can be expected both in the near-term and in the long-term, with most advancements in the analysis of real-time software coming in the long-term.

Prospects for the near-term

The advancements in the analysis of concurrent software that can be expected in the near-term will be the result of critically evaluating, fine-tuning, and integrating existing analysis techniques.

The theoretical limitations of general analysis techniques for concurrent software are known to be quite severe. The existence of theoretical limitations, however, is not surprising, as many questions about the behavior of general concurrent (and sequential) systems are known to be undecidable. Practical considerations, therefore, should be more important than theoretical ones for determining the usefulness of a particular analysis technique. A concurrent software analysis technique should be judged by the range of problems to which it can be practically applied, and not solely by its theoretical limitations. Substantial experience with a technique is required to make such a judgement. Experience with existing analysis techniques, however, has been stymied by a lack of automated support. It is therefore necessary to automate and experimentally evaluate existing analysis techniques.

The strengths and weaknesses of different approaches to validation of concurrent software have some complementary aspects. Testing, for example, does not require any simplifying assumptions, but suffers from a lack of breadth. Formal analysis, on the other hand, may establish properties of entire classes of execution sequences. The associated formal models, however, tend to be overly simplified and the analysis, itself, difficult and error-prone. By balancing the strengths and weaknesses of different approaches to software validation and verification, it should be possible to obtain techniques that can be more widely and practically applied than those associated with a single approach.

The proolems that software developers have with using the mathematical formalisms required by various analysis techniques must also be addressed. Ideally, the underlying formalism should be transparent to the developer, who should be able to work in whatever notation is most appropriate for a system and its current state of development. Mathematical models required for analysis should be mechanically derived from the actual system descriptions produced during development (i.e, specifications, designs, and implementations). The analysis of the models, of course, could be performed by specialists who are trained in using sophisticated mathematical formalisms.

In summary, specific research and development activities that would facilitate near-term advancements in the analysis of concurrent software are:

- automation of existing concurrent system analysis techniques
- experimental evaluation of existing techniques
- integration of different approaches to validation
- design of appropriate interfaces between tools and developers

Prospects for the long-term

In the long-term, development and refinement of the mathematical foundations of concurrent and real-time computations can be expected to produce the most dramatic achievements. Model development is needed to advance the state-of-the-art both in formal analysis and verification. It will be necessary to formally verify concurrent real-time software if the high degree of reliability required for SDS software is ever to be achieved.

The mathematics of concurrency is just beginning to be understood. Several formal modeling schemes, supporting different types of analysis, have been developed. As described above, however, many of these models are too abstract to be useful except in the early stages of development. Formalisms that support verification and analysis at every stage of development are needed. Such formalisms should be compositional, for use with hierarchical development methodologies, and tolerate incompleteness.

The mathematics of concurrency has been studied much more extensively than the mathematics of time dependent computations. Consequently, existing analysis techniques for real-time software are ad hoc and primitive compared to those for concurrent software. This is an area that clearly requires extensive investigation.

In summary, specific research activities that are needed for long-term advancements in the formal analysis and verification of concurrent real-time software are:

- development of mathematical formalisms supporting verification of concurrent software
- development of mathematical formalisms supporting both analysis and verification of real-time software
- development of techniques for analyzing mathematical models of real-time software
- research in verification of concurrent and real-time software

4. Current Research at UCSB

Introduction

The need for highly reliable software is evident, particularly in the area of secure systems and life-critical applications, such as patient monitoring and air traffic control. In order to achieve reliable software it is necessary to validate that high order language code is consistent with its requirements. Currently there are two approaches to validating software systems: testing and formal verification. These are often considered to be opposing approaches, but they are complementary and can benefit from each other.

For the past six years the Reliable Software Group at UCSB, under the direction of Prof. Kemmerer has addressed the need for better languages and tools for designing, building, and validating software systems. In particular, the Reliable Software Group has designed and implemented a language for formally specifying and verifying software systems, a system for formally specifying, testing and verifying Pascal programs, and a system for testing formal specifications. These are called respectively ASLAN, UNISEX and Inatest. All three of these systems are implemented on a UNIX operating system running on VAXs and SUN workstations. The UNISEX system has been distributed and is being used in a number of universities and software development industries in the USA, Europe, and Japan. The ASLAN system has been distributed in the USA and Europe and the Inatest system is not yet ready for distribution.

In addition to developing software tools, during the last few years the Reliable Software Group has been studying the problems raised by concurrent and real-time programs. A formal language for specifying real-time systems has been designed. It is an extension of the ASLAN specification language and is called RT-ASLAN. The group has also looked at methods for extending symbolic execution techniques for verifying programs written in a tasking subset of Ada [DOD83].

A formalism for modeling the behaviors of concurrent programs has become an additional focus of research since Prof. Dillon joined the Reliable Software Group in 1986. The formalism, called constrained expressions, supports analysis of important safety properties of concurrent programs.

The group has also been investigating an approach to analyzing encryption protocols using machine aided formal verification techniques.

ASLAN

The ASLAN verification system [AK85] is an integrated system for designing, specifying, and verifying software systems. It is built around the ASLAN specification language. The ASLAN language is a nonprocedural assertion language that is an extension of first order predicate calculus with equality. It provides the user with a good human interface and deals explicitly with exceptions, which were the two primary goals for this language. The language is used to model systems as state machines. Key elements of the language are types, constants, variables, definitions, initial conditions, invariants, constraints, state transitions, levels, and mappings.

The proof methodology used in the ASLAN system is an extension of the Alphard [Sha81] approach to the correctness of data representations that handles more than one level of refinement and deals explicitly with exceptions. This methodology enforces rigorous connections between successive stages of development.

When using the ASLAN methodology to design a system one states the correctness requirements for the system as well as the top level design specification using the ASLAN language. The ASLAN specification processor then reads the specifications written in ASLAN and produces the necessary correctness conjectures to prove that the top level specification is consistent with the correctness requirements. The design specification is repeatedly refined to include more detail until a program design specification is derived. Each of the intermediate design specifications as well as the program design specification are verified to be consistent with the preceding level as the refinement process is carried out. By verifying that the top level design specification is consistent with the critical requirements and that each of the refined specifications is consistent with the preceding specification the lowest level specification is shown to satisfy the correctness requirements for the system. Furthermore, by verifying the specifications at each stage of the design process errors are detected immediately rather than after the system has been implemented.

UNISEX

UNISEX is a UNIX-based symbolic executor for Pascal [EK83, KE85, SKE83]. The UNISEX system provides an environment for both testing and formally verifying Pascal programs. The system supports a large subset of Pascal, runs on UNIX and provides the user with a variety of debugging features to help in the difficult task of program validation.

The testing technique implemented by UNISEX is symbolic execution, which uses algebraic symbols to represent the input values rather than using numeric or character inputs. These symbols are then manipulated by the program. By placing restrictions on the values that each symbol may represent the symbolic input represents a class of input values rather than a single value. That is, by properly restricting the input values each symbolic execution of a program can correspond to executing the program on a particular subdomain. When execution is completed the symbolic values obtained by the variables are analyzed to determine if they meet the programmer's expectations. This method of testing is known as symbolic execution and the tool that performs the symbolic execution is called a symbolic executor.

A symbolic executor can also be used to generate the necessary verification conditions that need to be proved to verify that a formally specified program is consistent with its entry and exit specifications. The type of correctness being verified is partial correctness as originally defined by Floyd [Flo67]. That is, if the entry assertion is true when the program is invoked and the program terminates, then the exit assertion will be true when the program halts. This differs from total correctness where in addition to proving the above, one is also required to prove that the program will terminate.

The UNISEX system is used in both of these ways to test and verify Pascal programs.

Inatest

As stated above, formal specification and verification techniques are used to increase the reliability of software systems. However, these approaches sometimes result in specifying systems that cannot be realized or that are not usable. Therefore, it is necessary to test specifications early in the software life cycle to guarantee a system that meets its critical requirements and that also provides the desired functionality.

The Reliable Software Group at UCSB has designed and implemented a symbolic execution tool called Inatest [EK84, EK85]. Inatest is an interactive tool for testing specifications early in the software life cycle to determine whether the functional requirements for the system been designed can be met. It provides a testing environment, which allows the user to use various modes of operation to test formal specification written in Ina Jo* [SAM86]. The user submits the formal specification to the tool which then allows the user to interactively direct the tool to symbolically execute specified transforms to test the functionality.

The need for a specification testing tool and a suggested design were outlined by Kemmerer in [Kem85]. In this paper a functional requirement was defined to consist of a start predicate, a sequence of transforms to be executed, and a desired resultant predicate. The Inatest tool provides the user with several methods for defining the start predicate. It may be read in from a file, keyed in from the terminal, or "default" may be specified, which assumes the specification's initial predicate as the start predicate. In a similar manner the sequence of transforms to be executed and the resultant predicate may be read

* Ina Jo is a trademark of the system development group of the Unisys Corporation.

from a file or keyed in. The user also has the option to choose the transform to execute next and the actual parameters to be used.

After executing a transform the user may display the current, start, or desired resultant predicate, list the available transforms, or list the specification being tested. The user may also change the predicate defining the current state by adding a predicate or by defining a new start predicate. The tool also provides a path command that allows the user to display the transforms that have been executed since the start predicate was defined as well as any assumptions that the user may have made along the way. The Inatest user may also save the current state of the execution for further execution at a later time. Saved states may be restored in any order without affecting the other saved states.

RT-ASLAN

RT-ASLAN is a formal language for specifying real-time systems [AK86]. It is an extension of the ASLAN specification language.

The complexity of writing correct programs increases with the introduction of parallelism and is further complicated with the introduction of real-time constraints. Similarly, the difficulty of specification and verification increases from sequential to parallel to real-time programs. Like sequential programs, real-time programs are judged against critical correctness conjectures. Real-time systems, however, also must meet critical performance deadlines. Therefore, the verification of real-time systems involves demonstrating that the specified system meets the performance deadlines in every case and, in particular, in the worst case.

The RT-ASLAN language supports the specification of parallel real-time processes through arbitrary levels of abstraction. RT-ASLAN allows concurrent processes to be specified as a collection of subspecifications. The class of system specifiable are loosely coupled systems communicating through formal interfaces. It is also assumed that systems specified in RT-ASLAN have a processor associated with each subspecification. Therefore, the systems that can be specified with RT-ASLAN are neither process restricted nor resource restricted.

From RT-ASLAN specifications, performance correctness conjectures are generated. These conjectures are logic statements whose proof guarantees the specification meets critical time bounds.

Symbolic Execution/Ada

The Reliable Software Group has explored methods for extending symbolic execution techniques to specify and verify concurrent programs. In an earlier feasibility study [Eck83] an approach to symbolically executing concurrent programs written in Gypsy [GDS84] was investigated. The approach outlined in this paper was based on the interleaving approach introduced by Brand and Joyner [BJ78].

Recent work by the Reliable Software Group has resulted in the design of UNISEX-like symbolic executor for a verifiable subset of Ada, which includes tasking [Har88]. The subset does not include shared variables; all communication between tasks is limited to the rendezvous mechanism. The symbolic executor uses the interleaving approach; the execution of component tasks are interleaved to model their concurrent execution [HK88].

An alternative to the interleaving approach is also being investigated. The alternative approach is based upon verifying tasks in isolation and then performing cooperation proofs [Dil88]. The group recently completed a study comparing the strengths and weaknesses of these two approaches. The results suggest that the isolation approach may be better suited for formal verification, while the interleaving approach may be useful as a basis for symbolic testing and detailed analysis [DKH88].

Constrained Expressions

Each behavior of a concurrent system can be viewed as determining a sequence of event occurrences. This sequence can be represented by a string of event symbols. The set of possible behaviors of a concurrent system thus determines a language over an event alphabet. A constrained expression representation of a concurrent system provides a finite representation for this (potentially infinite) language.

Constrained expressions are usually generated by deriving them from a system description in some other notation. They can be mechanically derived from descriptions in a wide variety of development notations [DAW88].

While a development notation is chosen to facilitate description and maximize expressive power, constrained expressions are designed to encode the information required for the analysis of important system properties. The constrained expression representation of a system provides a formal basis for arguments concerning the order and number of occurrences of particular events in behaviors of the system. Arguments of this nature have been widely applied for analyzing concurrent systems for such properties as mutual exclusion, deadlock, and starvation. The constrained expression formalism thus offers a general intermediate form that both supports formal analysis and allows development in the appropriate notations. The application of the constrained expression analysis techniques in a realistic distributed software development setting is illustrated in [ADW86].

The Reliable Software Group is currently developing a prototype toolset for the analysis of concurrent program designs based on the constrained expression formalism. It is targeted for use with an Ada-like design language, called CEDL [Dil86]. The toolset will contain a deriver, simplifier, behavior generator and various analysis tools. The deriver produces a constrained expression representation for a design from its CEDL description. These constrained expressions are then optimized by the simplifier. This optimization is based on specific features of the given design. The behavior generator helps a designer produce example behaviors from a constrained expression representation. The behavior generator is particularly useful for interpreting the results of the analysis tools and for testing a design. Finally, the analysis tools automate the algebraic analysis techniques described in [AW85, ADW86].

Using Formal Verification Techniques to Analyze Encryption Protocols

The approach used by the Reliable Software Group to analyze encryption protocols is to formally specify the components of the cryptographic facility and the associated cryptographic operations. The components are represented as state constants and variables, and the operations are represented as state transitions. The desirable properties that the protocol is to preserve are expressed as state invariants and the theorems that need to be proved to guarantee that the system satisfies the invariants are automatically generated by the verification system.

This approach does not attempt to prove anything about the strength of the encryption algorithms being used. On the contrary, it assumes that the obvious desirable properties, such as that no key will coincidentally decrypt text encrypted using a different key, hold for the encryption scheme being employed. Axioms are used to represent the properties that the encryption algorithms are to satisfy. Thus, by replacing the current axioms with axioms that express the properties of a different encryption scheme one can verify the system assuming the use of the new encryption scheme.

This approach was used on a single-domain communication system using dynamically generated primary keys and two host master keys, as described in [MM88]. The system was specified in Ina Jo and the

Inatest system was used to analyze the specification. This analysis revealed a weakness in the encryption system. Another weakness that occurs when using DES semi-weak key pairs was also discovered using this approach. The specification and more details on the analysis can be found in [Kem87].

Future Directions

The Reliable Software Group plans to continue building and distributing software tools to be used in the development of reliable software systems, and to enhance the current set of tools making them useful for a wider domain of problems and providing a better user interface.

ASLAN/RT-ASLAN

Extensions to the ASLAN language processor to accommodate RT-ASLAN are being designed and will soon be implemented. For the future, low (almost implementation) levels of specification must be investigated more closely. It is easy to say "communicating transitions call interface transition," or that "TIME is an implicit variable representing a clock"; however, these things may be hard to implement in a way that is consistent with the RT-ASLAN formalisms. Provisions must also be made for global invariants governing the interaction of communicating processes and the interface processes.

Further research will extend the RT-ASLAN language and proof methodology to allow the formal specification and verification of a less restricted class of real-time systems.

Symbolic Execution/Ada

As stated above, the Reliable Software Group has recently designed an approach to symbolically executing concurrent programs written in a verifiable tasking subset of Ada. Two different models of symbolic execution were investigated; one based upon interleaving the task components [HK88] and the other based upon verifying the tasks in isolation and then performing cooperation proofs [Dill88].

Preliminary research indicates that the isolation approach is better-suited to formal verification. Further research is required, however, to determine if the approach could be integrated into a more general verification methodology supporting data abstraction, such as RT-ASLAN, and whether implementation of the interleaving approach as an analysis tool would be a worthwhile effort.

In addition, the existing proof rules do not deal with timing, and, therefore, cannot be used for verifying timing requirements for Ada programs. Further research will define proof rules for Ada constructs that use timing, and these will be incorporated into the current symbolic executor design.

Constrained Expressions

Experience with using constrained expressions for analysis has been limited to small-scale "toy" programs due to a lack of automated support. The constrained expression toolset, which is currently under development, will allow experimental evaluation of the constrained expression analysis techniques. The Reliable Software Group will investigate how well the constrained expression analysis techniques scale-up for use with more complex software systems.

The use of constrained expressions for modeling real-time systems is also being explored. Preliminary research indicates that the algebraic analysis techniques can be applied to a constrained expression representation of a real-time system to determine upper bounds on the elapsed time between specified event occurrences.

Inatest

The current version of the Inatest system includes only minimal simplification and theorem proving capabilities. Therefore, the major thrust of the research on this project is concentrating on designing, implementing, and testing efficient simplification and decision procedures.

Analyzing Encryption Protocols

The group will continue to investigate the usefulness of applying verification techniques to encryption protocols. The approach will be applied to other encryption protocols and an attempt will be made to identify the class of protocols that are amenable to this technique.

In addition, enhancements to the group's software tools that might aid in the analysis of encryption protocols will be identified.

References

[AK85] B. Auernheimer and R.A. Kemmerer, "ASLAN user's manual", Department of Computer Science, University of California, Santa Barbara, TRCS84-10, March 1985.

[AK86] B. Auernheimer and R.A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986.

[ADW86] Avrunin, G.S., L.K. Dillon, and J.C. Wileden, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986.

[AW85] Avrunin, G.S., and J.C. Wileden, "Describing and Analyzing Distributed System Designs," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985.

[BJ78] Brand, D., and W.H. Joyner, "Verification of Protocols Using Symbolic Execution," *Computer Networks*, Vol. 2, 1978.

[DAW88] L.K. Dillon, Avrunin, G.S., and J.C. Wileden, "Constrained Expressions: Toward Broad Applicability of Analysis for Distributed Software Systems", to appear in ACM Transactions on Programming Language and Systems, 1988.

[DKH88] Dillon, L.K., R.A. Kemmerer, and L.J. Harrison, "An Experience with Two Symbolic Execution-based Approaches to Formal Verification of Ada Tasking Programs", submitted to second workshop of Software Testing, Verification, and Analysis, Banff, Alberta, Canada, July

[DOD83] "Reference Manual for the Ada Programming Language," ANSI/MIL-STD- 1815A-1983 United States Department of Defense 1983.

[Dil86] Dillon, L.K., "Overview of the Constrained Expression Design Language," Technical Report TRCS86-21, Computer Science Department, University of California, Santa Barbara, October 1986.

[Dil88] Dillon, L.K., "Symbolic Execution of Ada Tasking Programs," *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, Medford, New Hampshire, May 1988.

[Eck83] Eckmann, S., "Symbolic Execution of Concurrent Programs in Gypsy," Computer Science Report, Dept. of Computer Science, University of California, Santa Barbara, California, March 1983.

[EK83] Eckmann, S. and R.A. Kemmerer, "A User's Manual for the UNISEX System," Dept. of Computer Science, University of California, Santa Barbara, California, December 1983, (revised April 1985).

[EK84] Eckmann, S. and R.A. Kemmerer, "Preliminary Inatest User's Manual," Dept. of Computer Science, University of California, Santa Barbara, California, April 1984.

[EK85] Eckmann, S. and R.A. Kemmerer, "INATEST: An Interactive Environment for Testing Formal Specifications," Third Workshop on Formal Verification, Pajaro Dunes, California, February 1985, also appeared in *Software Engineering Notes*, Vol. 10, No. 4, August 1985.

[Flo67] Floyd, R., "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics, American Mathematical Society, Providence, Rhode Island, 1967.

[GDS84] Good, D.I., B.L. DiVito, and M.K. Smith, "Using the Gypsy methodology," Institute for Computing Science, University of Texas, Austin, Texas, June 1984.

[Har88] Harrison, L.J., *CASEX: A Concurrent Ada Symbolic Executor*, Master's Thesis, Department of Computer Science, University of California, Santa Barbara, California, June 1988.

[How75] Howden, W.E, "Methodology for the generation of program test data." *IEEE Transactions on Computers*, c-24(5):554–215, May 1975.

[HK88] Harrison, L.J. and R.A. Kemmerer, "An Interleaving Approach for the Symbolic Execution of Ada Tasking Programs," *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, Medford, New Hampshire, May 1988.

[Kem85] Kemmerer, R.A., "Testing Formal Specifications to Detect Design Errors," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January 1985.

[Kem87] Kemmerer, R.A., "Analyzing Encryption Protocols Using Formal Verification Techniques," *Proceedings of CRYPTO 87*, Santa Barbara, California, August 1987.

[KE85] Kemmerer, R.A. and S.T. Eckmann, "UNISEX: A UNIx-based Symbolic EXecutor for Pascal," *Software Practice and Experience*, Vol. 15, No. 5, May 1985.

[Mil80] Milner, R., *A Calculus of Communicating Systems. Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[MM80] Meyer, Carl H., and Stephen M. Matyas, *Cryptography*, John Wiley, 1980.

[MZGT85] Mandrioli, D., R. Zicari, C. Ghezzi, and F. Tisato, "Modeling the Ada task system by Petri nets." *Comp. Lang.*, 10(1):43–61, 1985.

[Pet77] Peterson, J., "Petri nets." *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Sha81] Shaw, M., ed., *ALPHARD: Form and Content*, Springer-Verlag, New York, 1981.

[Stu77] Stucki, L., *New Directions in Automated Tools for Improving Software Quality.* Volume 2, Prentice Hall, Englewood Cliffs, 1977.

[SAM86] Scheid, J., S. Anderson, R. Martin, and S. Holtsberg, "The Ina Jo Specification Language Reference Manual," SDC document, System Development Corporation, Santa Monica, California, January 1986.

[SDL78] Sayward, F., R. DeMillo, and R. Lipton, "Program mutation: A new approach to program testing." In *Proceedings of Software Specification and Testing Technology Conference*, Office of Naval Research, April 1978.

[SKE83] Solis, D.M., R.A. Kemmerer, and S.T. Eckmann, "UNISEX Pascal Language Reference Manual," Dept. of Computer Science, University of California, Santa Barbara, California, December 1983, (revised April 1985).

[Tay83] Taylor, R.N., "A general-purpose algorithm for analyzing concurrent programs." *Communications of the ACM*, 6(5):362–376, May 1983.

[ZW81] Zeil, S.J., and L.J. White, "Sufficient test sets for path analysis testing strategies." In *Proceedings of the 5th International Conference on Software Engineering*, pages 184–191, March 1981.

Dr. William Howden

Dept. of Computer Science and Engineering
University of California
Mail Code C-014
La Jolla, CA 92093-0114

CRITICAL GAPS IN VALIDATION TECHNOLOGY

1. **Completeness.** Perhaps the worst problem in testing is that after you have completed a prescribed set of validation activities, there is no clear specification of what has been achieved. What does it mean to test all branches? Why, and in what way does this make software more reliable? Another way of putting this is "You don't know what you know." Without a comprehensive way of viewing validation methodology no meaningful standards can be established.

2. Operational environment profiles. System testing requires that a system be tested in the same way that it will be used in practice. When this is impractical, well-defined models of the operational model must be developed. These can range from input distributions for simple computational programs to complex state models for concurrent systems which interact with their environment in real time. In order to derive formal conclusions about the expected reliability of a system it is necessary to formally describe the operational profiles on which the conclusions are based.

3. Automation. There are several practical, effective methods for which there are no available tools. This situation exists for a variety of reasons. Those organizations which have resources often do not think it is important. Those that think it is important - university and institute research groups do not have the resources. The availability of funds for the development and analysis of tools for current methods would help to solve this.

4. Early validation. Everyone agrees that early validation reduces costs and is critical to effective software development. This topic is not well-developed. For example, what is the relationship between early validation and traditional methods which analyze code? A big problem is that there is no general model for validation at any stage of development so there is no notion of what a complete set of early validation activities would consist of. This and problem one are potentially solvable using the process error models in the Faultless Software project at UCSD.

CURRENT VALIDATION RESEARCH AND DEVELOPMENT ACTIVITIES

I recently completed an extensive project on fault-based testing. The goal of this work was to categorize fault classes and to develop testing rules for choosing test data which would probably reveal the presence of those faults. This work could be used as the basis of an advanced test coverage tool. It would not only keep track of whether or not a program component had been executed, but whether it had been executed over data which would reveal any common faults that it might contain. All of this work is summarized in my book *Functional Program Testing and Analysis* which was published by McGraw Hill in the spring of 1987.

The main organizing idea in my book is that programmers develop software by synthesizing functions. Errors result in faults in the structures which are used to join old functions together to make new functions. There are two basic ways of thinking about programs: as hierarchical top down functional structures and as horizontal state transition diagrams. The hierarchical approach is used in structured programming and views programs as structures in which one function consists of a structure of embedded functions. The state approach views programs as structures in which functions are used to transfer one state into another. This approach to the discussion of validation is comprehensive and makes it possible to discuss and compare diverse program validation methods. Completeness of testing is achieved by having a general model of how software is constructed and of the reasoning errors that humans make during this process, by identifying the fault effects of these errors, and by then constructing methods for detecting these faults.

Since completing the book I have developed the basics of a more general error model which is applicable to all software products and not just code. It is based on the idea that humans reason about complex objects using abstraction and decomposition. Different validation techniques are suitable for different kinds of abstraction and decomposition errors, and the approach can be used to relate methods as diverse as fault-based testing, proofs of correctness and static analysis.

The goal in all of this research has been to develop practical, effective validation methods and tools. But not a scattered collection of methods for whom there is well-developed justification. There had to be some overall picture of what a complete approach to validation would be, how a proposed method fitted into this picture, and why it was effective for the particular class of problems for which it was designed. Both the program oriented model in my book, and the more general model which has been recently developed, provide this overall picture.

In an early empirical study I carried out of a large 100k system, I discovered that the problems of big systems were not so much errors in computations, as errors in keeping track of data, the current status of devices, and in using information prepared and maintained in some part of the program other than the one currently being worked on. My research is sponsored by the Navy, and the real-time avionics systems which have become the sub-area in which I am working have this property. For this reason, I discontinued work on fault-based testing methods for computational faults after finishing my book and went back to analyzing large system problems. The result of this has been the development of a technology called flavor analysis, and the construction of a recently completed tool for doing flavor analysis of the avionics on the AV8B (Marine Corps Harrier). Flavor analysis fits into the bigger picture as a method for detecting decomposition errors.

The basic idea in flavor analysis is that programmers make false assumptions when they are working in one part of the system about some other part of the system. These assumptions can involve problems such as missing code or wrong variable names. The occurrence of these errors is detected by including information in the code about assumptions, in the form of flavor statements, which are like a kind of dynamic type, and by then analyzing flavor statements for consistency. There are two types of flavor statements. One summarizes current assumptions about flavors, and is like a kind of abstract incremental assertion, and the other summarizes the flavor effects of statements or sections of code. Flavor statements also serve as documentation comments and can be translated into both compile time and run time built-in program checks. Their main use is to allow static consistency analysis for the detection of false assumption decomposition errors.

The AV8B flavor analyzer uses a very simple notion of flavor analysis. A more complex version, which uses *flavor logic*, is now being developed. The new method also allows users to document and check assumptions about scheduling and other control activities. In addition, it contains rudimentary temporal logic capabilities for specifying and checking information which is either "always" or "sometimes" true. It is going to be used to validate real-time and concurrency properties of the interacting tasks in the AV8B avionics.

Both the simple flavor analysis tool which has recently been completed, and the newer analyzer, are meant to be real tools to be in production use within the next one or two years. Their initial area of application is the AV8B avionics. This work is being carried out as part of the Faultless Software Project at UCSD.

REFERENCES

[Howd87] Howden, W.E. 1987. *Functional Testing and Analysis*. McGraw-Hill.

[Howd88] Howden, W.E. 1988. *Comments Analysis and Programming Errors*. University of California at San Diego. Technical Report Cs88-142.

[Howd89] Howden, W.E. 1989. *Verifying Programs without Specifications*. University of California at San Diego.

METHODS AND TOOLS FOR INCREASING RELIABILITY OF EMBEDDED ADA SYSTEMS

Timothy E. Lindquist

Arizona State University
Computer Science Department
Tempe, Arizona 85287

1. INTRODUCTION

The software community is aware that the required reliability of a system (and its software) ranges depending upon the critical nature of the application. Tailored management practices along with special methodologies and system analysis techniques must be employed for critical reliability requirements. In the same sense, using certain development methods and even using certain languages and language constructs can affect the reliability of a system. For Ada, as the case is made below, system oriented facilities present particular challenges to the software engineer. Here to, special analysis techniques must be employed for critical applications.

Further, current Ada use has indicated a shift from the norm in development efforts for design, code and test activities. We are noting claims that less testing effort is required as the result of more front-end effort in design. Many view this as a positive sign regarding Ada's suitability. For reliability, however, these findings are most likely negative. One certainly expects that, due to the informal nature of design objects, that analysis of designs falls behind the current state in code analysis techniques. To further complicate the issue, one cannot expect the mapping from design to code to be implicitly correct.

1.1 Constructs: Ada's Strengths are Reliability Weaknesses

One area of major concern when considering the use of Ada in applications with critical reliability needs, is the use of Ada's system oriented facilities. Once embedded targets accommodate Ada's capacity and performance requirements, applications will remove restrictions on the use of facilities such as dynamic allocation, tasking and exceptions. This will be a step forward for most aspects of software engineering. When using these constructs at the application level, however, aspects of reliability relating to them moves from the executive or operating system domain to the application domain. To further complicate the problem, several planned uses of the language by the government involve distribution.

1.2 Test Generation and a Model for Test Administration

One approach to solving the reliability problem for critical software is to apply several analysis techniques. While doing so would certainly cause duplication of effort and redundant analysis, the benefits in reliability overshadows duplicity. In the short term, one approach to minimize the impact on development and maintenance cost is to develop a single unifying model to represent test generation and test administration. Such a model would be equally applicable to various analysis techniques.

An Ada package defines a software interface between the package user(s) and the objects manipulated by the package. Interface syntax is defined by the procedures in the package specification, and interface semantics is defined by the body of the package. The software-to-software interface provided by packages can provide the basis for an object-oriented software engineering environment. Tools in such an environment constitute the operations available on software engineering environment objects. An entity management system allows cooperation among tools, and a rich typing mechanism including specialization and generalization provides the capability to increase the reusability of tools and to decrease the complexity of highly integrated tools. Testing tools can take advantage of these interfacing mechanisms to allow diverse approaches to test case generation to fit into a single automated toolset. Various test case selection methods exist including:

1. functional testing,
2. dynamic and static testing
3. domain testing
4. mutation analysis
5. path analysis
6. data dependency analysis

In addition to these, various verification and evaluation methods exist. Research has shown that each approach has both strengths and weaknesses. Different aspects of reliability are enhanced by different methods. In the short term, commonalities among approaches can be found by looking at the test objectives produced by each method, and how the method fits into life-cycle activities. The potential exists, in the short term, for a unifying test administration model and a tool to separately accommodate each of the test selection criteria listed above. We believe that such as model should be built on an entity-relationship based object management system (the Common APSE Interface Set is an appropriate starting point for such a system). The unifying model should allow as many techniques as possible to share common program representations, specifications and test drivers. Further, the model should accommodate a common format for test plans, test objectives, test scenarios, test data and test programs.

The long term goal of reliable software can best be met by achieving the equivalent of a calculus of program testing. Such a calculus would have a formal method as its basis. The formal method must be able to provide a semantic description of the language with the ability to derive formal properties of programs. The method must also lend itself to generating test cases based on well selected subdomains of a program's space. The criteria for selecting the subdomains is continuity of program behavior in that domain.

At Arizona State we are addressing topics related to these needs using a technique we've derived from symbolic execution:

1. Embedding system test generation,
2. A framework for test administration,
3. An Intelligent Debugger for Tasked Ada Programs (AdaTad), and
4. A methodology for testing tasked software.

A pictorial view of these topics representing the relationships among them is shown in Figure 1. Testing begins with test generation and administration. Tests are identified and relationships among the underlying specifications, requirements and code are recorded in an appropriate framework. Tests are conducted using a methodology specifically designed for software involving tasking. The methodology provides for increased attention to system-wide characteristics, and is implemented as an interactive test assistant. The test assistant produces test scenarios which serve as input to both an Ada task debugger (AdaTad) and test administration. The debugger executes test scenarios using its connection to the embedded application's execution environment.

The remaining sections of this paper detail the tool developments that we are currently undertaking; the first task investigates and implements a model of test administration that may be applied to any of a number of test identification methods. This task also includes the continued development of a test case generation tool already begun at Arizona State University [Lin-88A]. This material is explained in Section 2. The second aspect of this research is the continued development of our methodology for testing tasked programs and the implementation of a prototype tool to support the methodology. As explained in Section 3, the methodology consists of three levels of testing with specific automated support for each level. These tasks work in conjunction with AdaTAD, an intelligent automated debugger for Ada that is under development at ASU as described in [Fai-86].

Our first research goal is to develop such an interface in the context of embedded systems testing and to demonstrate its utility. We will employ the entity-relationship based software which has already been developed at ASU as a part of the Common APSE Interface Set (CAIS) Operational Definition project. With the help of Ada Joint Program Office funding, we have developed an executable version of CAIS, and we have extended it to include a typing mechanism as defined in [Lin-87]. We also plan to continue our work in test case generation in the areas of tasking and exception mechanisms [Lin-88B]. This work will continue the development of the Test Generator (IOGen and Testgen). We already have a prototype, consisting of two major components. The first is a test predicate generator, as developed by Jenkins [Jen-86], called IOGen. Test predicates for Ada are identified by a technique derived from symbolic execution. The second component is a tool to support interactively converting predicates into test objectives [Cof-87], called TestGen. In addition to expanding the features of Ada recognized, our prototype has suggested an alternative implementation approach, and method for identifying more complete test sets.

2. TEST CASE GENERATION AND ADMINISTRATION

One of the Primary advances of the Ada program, which stands independently of the technical merits of the language, is the existence and administration of the ACVC (Ada Compiler Validation Capability) and the development of the ACEC (Ada Compiler Evaluation Capability). The ACVC is a suite of from 2600 to 4000 tests (depending on the version) aimed at assuring a compiler's adherence to the standard. Often, users of validated compilers note deficiencies in the suite. There is, however, already no doubt that Ada has demonstrated a much higher degree of transportability than achieved by other languages. For instance, Texas Instruments has reported on an effort to transport 20,000 lines of the AIM (Ada Interactive Monitor) from the Data General implementation to the Digital Equipment implementation. They report a total transportability effort of 2 man months. A large portion of that effort was devoted to converting system level calls. Additionally, the authors have transported 18,000 lines of Ada code from VAX/VMS running DEC Ada to a SUN 3/50 running Verdix Ada in one half of a man month. All users of the ACVC realize its incompleteness with respect to verifying a compiler's consistency with the language reference manual. Nonetheless, ACVC has been tremendously successful partly due to its technical content and partly due to its administration. It is this combination of technical viability and administrative capability that motivates our investigation of a framework for testing embedded Ada software.

Along with System V UNIX, the Ada Program is also leading the way in developing a capability similar to the ACVC for kernel (or operating system) services. Currently under development is a validation capability for the CAIS (Common Ada Programming Support Environment Interface Set), which is a government standard (MIL-STD-1838) kernel interface to support Ada software

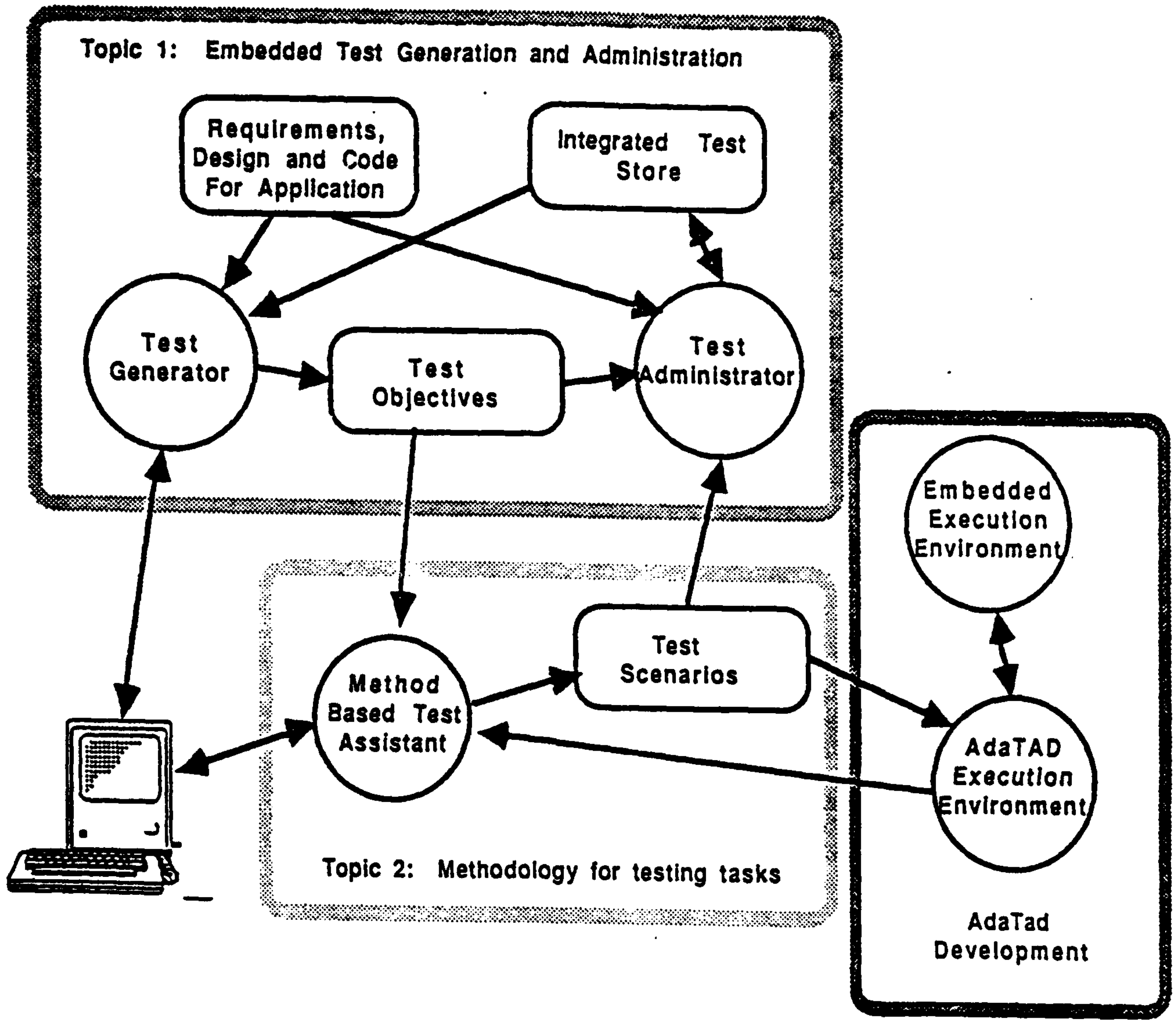
Topic 1: Embedded Test Generation and Administration
Requirements, Design and Code For Application
Integrated Test Store
Test Generator
Test Objectives
Test Administrator
Embedded Execution Environment
Method Based Test Assistant
Test Scenarios
AdaTAD Execution Environment
Topic 2: Methodology for testing tasks
AdaTad Development

Figure 1. ASU Ada Test Project Overview

engineering tools. The philosophy underlying the suite is that development tools will be more transportable if they are written to depend on a set of services that are common among hosting environments. The CAIS is founded on an entity-relationship approach to the store of information needed in software engineering environments. At Arizona State University, the authors have implemented an operational version of CAIS in Ada, called CAISOD. This set of interfaces is currently being extended to incorporate an object-oriented typing mechanism as reported by Lindquist [Lin-87]. This tool will provide the basis for constructing our Test Administrator.

2.1 Generating Tests

The method designed by Lindquist and Facemire [Lin-85] to construct validation tests for the CAIS utilizes an Ada-like Abstract Machine description of CAIS, which is assumed to be correct with respect to the specifications. Using a technique derived from symbolic execution, we analyze the abstract machine description and identify tests in a White-box fashion. Test predicates are formed by traversing the symbolic execution trees and referencing the specifications. The predicates are in the form of input/output pairs describing the input to a test and its expected result. Predicates can be converted into test programs that can be administered in a Black-box fashion.

The prototype of this validation system, as reported in [Lin-88A], is shown in Figure 2. IOGen [Jen-86] isolates predicates from Ada code. The pairs are then analyzed by the Test Generator [Cof-87], TESTGEN, to select specific pairs for testing, to remove implementation details, and to formulate the validation test programs. Each execution path through the program is identified by a path through the tree. Each path through the tree has one or more predicates associated with it, which identifies conditions causing the path to be executed. In Ada there may be multiple conditions along a path reflecting exception generating conditions and condition coverage of predicates encountered along the path. Actions taken along a path describe the outputs that are generated by executing the path. Jenkins [Jen-86] describes a preliminary implementation of IOGen.

We are currently in the process of moving IOGen onto an LALR(1) based recognizer. This allows us to expand the subset of the language processed by IOGen. We have constructed a set of compiler-compiler tools (Alex and Ayacc) which are written in Ada and produce Ada tables and Ada semantic action calls. We are currently converting IOGen to run under these tools. We have also begun expanding our test case generation method to accommodate the real time aspects of Ada. Of particular interest thus far have been exceptions and tasking [Lin-88B].

2.2 Test Administration

We also plan to convert IOGen and Testgen to produce test objectives that are based on requirements, functional specifications and design information as well as the code itself. To do so, we have found that an integrated store of requirements, specification, design and test information must be available to the test generator. The integrated store retains the underlying relationships among units of information in different documentation. For example, if a given requirement drives one or more functions in the specification, then these relationships should be represented. We plan to explore this store and to build a prototype framework for it on top of the CAIS Operational Definition. We have already implemented an inheritance based typing mechanism on top of CAIS [Lev-88]. The addition will allow us to explore definitions of objects representing products of software life-cycle activities. Using CAIS, we can represent relations between the objects and test objectives that would be produced by a testing method.

In summary, we have chosen to continue using a testing method based on symbolic execution. Our experience has shown that it can made to produce a thorough set of tests and it has the added advantage that its basis is a program verification method. To provide an appropriate formal basis for testing, we must move towards a "calculus" of test case selection, and in that regard, symbolic execution's origin as a verification method is significant. In transitioning to test generation based on more than just the source code, we've found that an integrated data store is needed. Such a store, if constructed appropriately, can serve as the information base for any number of test generation methods.

3. A METHODOLOGY FOR TESTING TASKED SOFTWARE

The applications for which Ada was developed require distributed implementations of the language and extensive use of tasking facilities. Testing technology as it applies to parallel features of languages currently falls short of needs. Thus, the development of embedded systems using Ada poses special challenges to the software engineer. Support for simulating distributed target machines, testing facilities for tasked programs, and debugging support applicable to simulated and to real targets

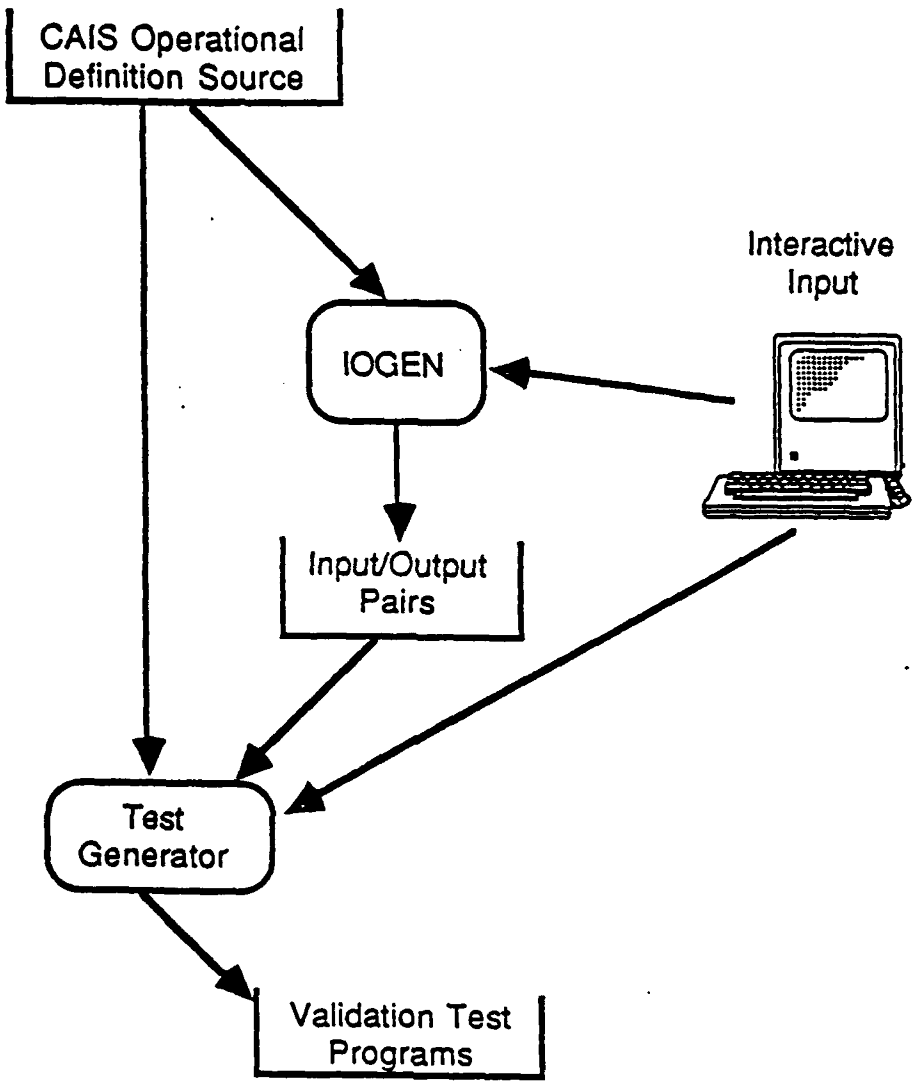

Figure 2. The Automated Assistant for Generating Validation Tests

all need to be addressed. Our work is formulating a technique for debugging Ada tasked programs based on the debugger AdaTAD. The underlying assumption in this work is that a close and well-defined tie must exist between: a technique to generate test cases for language facilities such as Ada tasking, a methodology for testing software that utilizes these facilities and debugger-like tools to support administering test cases employing the methodology.

3.1 Problems with Testing Tasked Programs

One view of program testing indicates that a program has been tested when every statement in the program has been executed at least once and every possible outcome of each program predicate has occurred at least once. Considerable literature addressing techniques for testing software reflects a view of testing that is consistent with this definition. Although this definition does not naturally extend to tasked programs, it represents the view that testing occurs late in software development and is oriented towards validation.

In contrast, debuggers have traditionally had utility in development activities, before validation of software begins. Accordingly, debuggers are used as automated support for locating errors (once their presence is known) and determining what is needed to correct errors. Ideally, testing is used to identify the presence of errors and debuggers to support location and correction. When tasking facilities are included in a language, however, the software designer is left without good testing techniques, and debugging must enter into the process of identifying the existence of errors. Helmbold [Hel-85] suggests that "Debuggers for parallel programs have to be more than passive information gatherers–they should automatically detect errors".

When tasking errors directly depend on the semantics of the language, a debugger is able to actively aid in detecting errors. More commonly, errors are also dependent on the specific logic of task interaction and the use of the language. To take an active role in identifying the more complex type of errors, the debugger must include facilities to analyze the logic of the program, to generate test cases and to aid in executing test cases. Helmbold distinguishes types of tasking errors as "Task Sequencing Errors" and "Deadness". AdaTAD provides task information that may be used to detect either type of tasking errors, although it does not actively detect errors. The efforts of this task area will address generation of thorough test cases for concurrent constructs.

"Task Sequencing Errors" are those that result when task communication and synchronization occurs in an unexpected manner. Ultimately, we recognize that the program under observation may to some extent be perturbed by the observer. This phenomenon is commonly known as the Heisenberg Uncertainty Principle. Although evident in all configurations, it is most pronounced in monitoring a distributed program in its actual target environment.

A methodology for debugging tasked Ada programs with AdaTAD are presented jointly with AdaTAD in [Fai-86]. Aside from this work, very little has been done to address this very difficult area of software engineering. One notable exception, however, is the work being done by Tai. Our approach to task debugging centers on removing task errors from three successive levels of consideration. Errors within tasks, which are principally independent of other tasks, are first addressed. Next, the communication and synchronization structure among tasks is addressed, and finally, application specific concerns are addressed. Facilities of AdaTAD are designed to specifically support debugging at these three levels. Further, AdaTAD has a user interface that is designed to graphically depict task interactions as demonstrated in Figure 3 and Figure 4.

The first level of usage for the debugger is to address logic errors within each of a programs tasks. These errors are exclusive of intertask communication and synchronization. Removing them is synonymous to removing errors detected during unit testing of software. At this level, we assume that interactions with other tasks are correct and examine the activities of the task itself. Testing and debugging at this stage considers a software module in absence of all elements of its environment except any procedures or functions it calls. For example, a task may use information obtained from other tasks to retrieve and update information in a database. Task logic to perform operations on the database is considered, at this level, exclusive of synchronization with other tasks.

After checking the logic within a task, the communication and synchronization among tasks is considered. This step is analogous to integration testing in that the cooperating among possibly several tasks is addressed. Data flow and control flow through tasks of the program are observed at this level of testing and debugging. From the perspective of a single task, this level checks, in a rudimentary manner, the task's tasking environment. Subtle timing interactions and interactions with the operating environment are left to the final level of checking. The final stage of debugging considers the operating

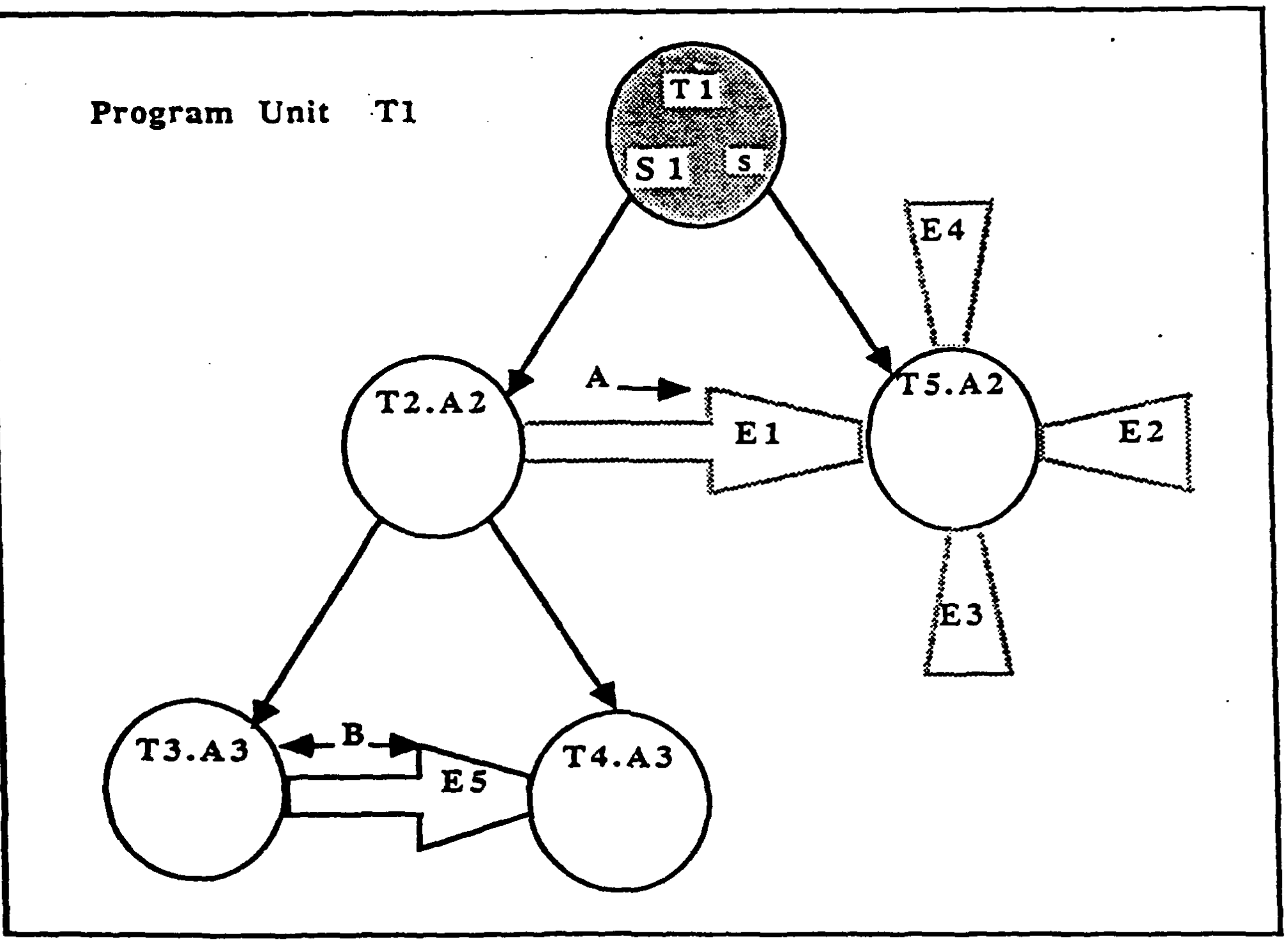

Figure 3. Task Interaction Status Window of AdaTAD

environment in which the tasks must execute. For an embedded system, this may include operating within a set of heterogeneous processors, each with different resources and capabilities.

Testing and debugging at this level is often accomplished with a simulation of the operating environment. While specific tools are necessary to support this activity, AdaTAD provides facilities that are useful in a general manner to the problem of addressing the operating environment. The problems that may arise in this phase of testing include timing inconsistencies among tasks, space requirements of a task, or resource contention caused by task interaction.

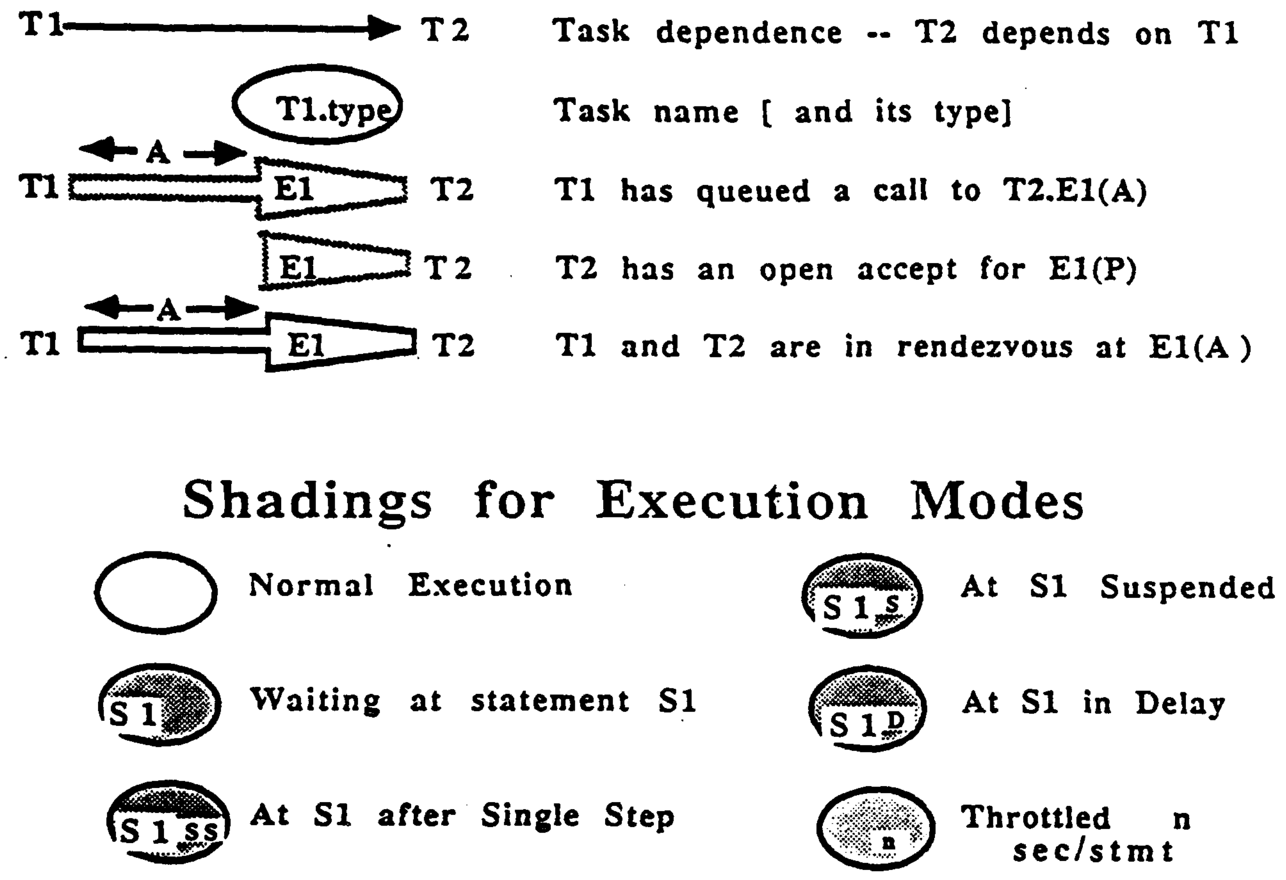
T1
T 2
Task dependence -- T2 depends on T1
T1.type
Task name [and its type]
A
T1
E1
T2
T1 has queued a call to T2.E1(A)
E1
T 2
T2 has an open accept for E1(P)
A
T1
E1
T2
T1 and T2 are in rendezvous at E1(A)
Shadings for Execution Modes
Normal Execution
S 1
Waiting at statement S1
S 1 SS
At S1 after Single Step
S 1 S
At S1 Suspended
S 1 D
At S1 in Delay
n
Throttled n sec/stmt

Figure 4. Legend for Task Interaction Status Window

REFERENCES

[Ada-83] "Ada Programming Language," ANSI/MIL-STD-1815A, January, 1983.

[Cof-87] Coffman, M.L. "Validation of Ada Package Interfaces," Masters Thesis Arizona State University Computer Science Department, 1987.

[Fai-85] Fainter, R. G. "AdaTAD-A Debugger for the Ada Multi-Task Environment," Ph. D. Dissertation, Department of Computer Science, Virginia Tech, Blacksburg, Virginia, June, 1985.

[Fai-86] Fainter, R. G. and Lindquist, "Debugging Tasks with AdaTAD," Proceedings of the First International Conference on Ada for the NASA Space Station.:, June, 1986, Houston, TX.

[Hel-85] Helmbold, D. and D. Luckham, "Debugging Ada Tasking Programs," IEEE Software, March 1985.

[Jen-86] Jenkins, J.R. "Automated Generation of Input/Output Pairs for the CAIS Validation Test Suite," Masters Thesis Arizona State University Computer Science Department, 1986.

[Lev-88] Levine, D.P., "Toward the Production and Application of an Environment Type Definition Processor," Masters Thesis Arizona State University Computer Science Department, 1988.

[Lin-85] Linguist, T.E. and J.L. Facemire, "Using an Ada-Based Abstract Machine Description of CAIS to Generate Validation Tests," ACM Washington Ada Symposium, March 1985.

[Lin-87] Lindquist, T.E., P.K. Lawlis, D.P. Levine, "Typing Information in a Software Engineering Environment," Proc. of the Sixth Intl Conf on Entity-Relationship Approach, November 1987.

[Lin-88A] Lindquist, T.E. and J.R. Jenkins, "Test-Case Generation with IOGen," IEEE Software, January 1988.

[Lin-88B] Lindquist, T.E., I.S. Kwon, V.L. Wood, "Test Case Generation for Ada Exceptions and Tasking," In preparation.

[Luc-85] Luckham, D.C. and F.W. von Henke, "An Overview of Anna, a Specification Language for Ada," IEEE Software, March 1985, pp. 9-22.

Languages, Techniques and Tools for System Specification and Validation

by

David C. Luckham
Program Analysis & Verification Group
Computer Systems Laboratory
Stanford University

1. Abstract

This is a brief position paper prepared for the IDA workshop on Validation, Verification, and Evaluation of Software. The paper takes the position that systems combining both hardware and software components should be considered. "System" is presumed to be a concurrent computational system consisting of both hardware and software components. Concurrent computation includes computations that may be distributed over a number of processors, either loosely or tightly coupled.

The paper focuses on the need to develop specification languages for systems, techniques for utilizing precise formal specifications in analysis of systems, and finally, the need for tools that automate those techniques.

2. Problems in Validation and Verification of Systems

As we all know, traditional methods of testing and debugging systems are still with us. And they are still the most used and useful approaches to the problem of getting a system to do what we thought it ought to. But these old techniques, which include simulation, testing and test data generation, and debugging when errors in fielded systems are reported, are unsatisfactory. In fact, we are all here at this workshop on Validation and Verification (V & V from now on) because we all agree that new approaches and tools are needed.

In the past 20 years, formal methods have been proposed as an alternative technology to testing. Although these methods were proposed mainly for software, there has been an increasing effort over the past ten years towards applying them also to hardware.

There have been two main trends in applying formal methods to programming — specifically, to program correctness. The first is the "transformational approach". The programmer develops an executable program in small, rather obvious steps, from a "high level" specification. Each step is to be justifiable either as the result of an equivalence-preserving transformation, or by a consistency proof. The path from specification to code must be trod, transformation by transformation, or proof step by proof step, and no mistakes on the way! The idea is not to write wrong programs at all.

The second approach is "program verification". A program is developed, no matter how, and then analyzed for consistency with its specifications. The amount of information (verification conditions) that then needs to be analyzed (proved or disproved) is often enormous, its relationship to the program unclear, and the ability of automated provers to handle it in real-time, less than satisfying. The idea is to prove a program correct, and thus eliminate the need to test it or debug it.

Of course, both of the previous two characterizations are a little unfair. Formal methods approaches to program correctness embody valid longterm research directions towards solving the problem. In the 1970's they certainly resulted in development of new automated capabilities of constructing and analyzing simple programs. These capabilities may in fact have applications in programming processes, and indeed may form part of the foundation of future programming environments. And the study of formal methods also led indirectly towards the development of annotation and specification languages.

But it is also true that the new formal methods required drastic departures from standard software development processes. These changes were not supported by powerful enough capabilities. Consequently, the new techniques appear to demand more of the programmer than they pay off. So they have not yet become part of everyday practice. The lesson to be learned, therefore, from the recent history of formal methods, is that programming processes will not change before powerful enough automated tools (capabilities) are available to support the change.

In brief summary, the situation in Verification and Validation of systems today is:

1. Test data generation, and testing methods still possess the problems they always have:[1]
   - lack of adequate test data to cover all cases;
   - lack of time to do complete testing;
   - lack of automated methods of recognizing errors.
2. Simulation is still full of holes:
   - lack of adequate test data to cover all cases;
   - lack of time to do adequate simulation;
   - lack of validation of simulators;
   - lack of modular software engineering methods in developing simulators and simulator components;
   - lack of automated methods of recognizing errors.
3. Formal Verification has still to bear practical fruit:
   - lack of powerful automated provers, capable of handling the kinds of proofs that human users expect to be handled;
   - lack of good software engineering in the integration of verifiers into programming environments: verification tools have not been engineered to fit with other programming environment tools such as debuggers, and semantic analyzers, so they cannot be applied usefully to small problems where they might be useful;
   - lack of modular software engineering methods in design and construction of verifiers, verifier components, and provers: this has resulted in lack of portability, much repetition of old efforts and little new progress;

1. Remember we are talking concurrent systems here.

- lack of automated and usable provers for concurrent aspects of programming languages.

None of the items on this list appears to be insurmountable. It is simply a list of some of the problems in V & V at the present time.

3. Outlook

Validation and verification of systems requires development of new techniques beyond what exists at present. This is particularly evident in the case of concurrent systems, especially in the absence of global time (i.e., distributed systems). But it is true also of sequential systems.

Paradoxically, such techniques must probably develop gradually from what currently exists. This is a belief based on various political and social factors in information processing technology, which we do not discuss here. Suffice it to say that immediate, radically novel developments are unlikely. Research and development must focus on an evolutionary approach to new technology rather than a revolutionary approach. Improvements to current development processes should be sought. But, certainly, a long-term "revolutionary goal" should be clearly in sight — doubtless everyone will have a different revolution in mind.

There are two aspects to developing new techniques.

1. *Developing a new technical capability.*

   An important basis for new capabilities in V & V is the development of formal, machine processable, specification languages, and tools supporting techniques based on those languages.

   In our own work, we are developing a capability to write formal annotations of systems in powerful specification languages, and check the runtime behavior of systems for consistency with annotations.

2. *Applying the capability.*

   Application of new capabilities based on new specification languages, is the process by which we promote an evolutionary approach to V & V. It is also a process by which we learn to design better specification languages, and better techniques for using them.

   In our own work, for example, we are now developing methods of using formal annotations to improve upon the usual, and the sometimes creative, steps undertaken in normal debugging tasks. This represents an evolution of old style testing and debugging into a more formal and precise methodology.

3.1 Feasible near-term goals

Let us suppose that we are considering systems developed and implemented in languages such as Ada and VHDL. Some possible near-term goals towards improving capabilities in V & V are the following:

- **Development of specification and annotation languages.**

Specification and annotation languages compatible with Ada and VHDL are already under development[1]. Other languages, supporting various specification methodologies such as VDM, or algebraic methods (Larch) are being developed. This area is, as we have postulated above, fundamental to improving V & V technology.

- **Specification engineering tools.**

  These are tools that are intended to help us develop specifications, and manipulate them in various ways so that we can build new specifications out of previous ones. The primary objective of these tools is to aid in determining that specifications possess certain properties — e.g., consistency. Quite clearly this is an area where automated proof methods should apply.

  It is reasonable to expect production quality tools that would give substantial aid in building and proving specifications of the order of quite complex Anna/Ada package specifications, or Larch traits, within three years.

- **Behavioral analysis tools based on formal specifications.**

  Such tools are to be used for prediction of system behavior prior to implementation. Some tools in this category are under development now, and could be engineered for general use in production environments for Ada within two years.

- **Debugging tools based on formal specifications.**

  Tools in this category allow the use of specifications in analyzing errors. This is particularly critical in concurrent and distributed systems, where current kinds of "debuggers" cannot be expected to be of much help. Some tools in this category are under development now, and should be useful in production environments for Ada within three years.

- **Self-checking systems, utilizing formal specifications.**

  Techniques to down-load runtime checking of specifications on separate processors from those on which a program is executing, are being developed. Some initial experiments are being performed at this time. The significance of this capability is that the runtime penalty for checking behavior of software against its specifications, may be reduced to the point where it is practical to check specifications on a permanent basis. In three years we might expect to see some self-checking Ada systems in which certain kinds of specifications, such as security specifications for databases, are checked permanently.

  Similar research in the area of design of hardware for self-test has been underway for some time.

- **Multi-processor simulators, and self-checking simulations.**

  This is an important area, attempting to solve some of the issues raised above about simulation. It is difficult. So far, multi-processor simulation studies based on general principles, have often

1. See Section 4.

showed a decline in speed compared with single processor versions. However, there is evidence that faster simulators can be constructed so as to schedule concurrent operations based on assumptions about the system being simulated — such assumptions supplied by the system designer. If this turns out to be generally applicable, we may see much faster simulators based on multi-processors in three years.

Typical simulators, such as those currently being developed for VHDL, need enhancements to check the simulation results. It is feasible in three years to expect such simulators to provide a self-check capability based on annotation language enhancements of the system modeling language. Simulator validation suites can also be constructed based on annotation languages.

- **Verifiers.**

  Our assessment of the automated verification situation is that the capability for useful applications already exists. The problem lies in developing those applications. We do not believe that there is any hope, in the short term, of automated consistency proofs of systems, especially concurrent systems, that would raise confidence in a system behaving as intended.

  More effort must be put into the software engineering of verifier components, and their integration into various software development processes. For example, in three years we could expect semantic analysis, as usually performed during compilation of Ada programs, to be enhanced by proof techniques for optimization of Ada runtime constraint checks. Extensions of this kind of application are numerous — e.g., consistency checking of various Ada PDL concepts, such as exception propagation annotations, to be processed as a compiler option.

3.2 Long-term possibilities

Let us suppose that we are considering systems designed and developed in specification languages that evolved from such languages as Ada and VHDL. In effect, we are trying to foresee the results of ten years of evolution in V & V technology, supposing that that evolution follows our current plans. We must allow for the fact that evolution is always slower than our most pessimistic expectations.

A short list of long-term accomplishments in V & V might look like this:

- **Development of very high level systems specification languages.**

  We believe that specification and design languages will evolve quite rapidly over ten years, together with powerful support tools. These should include graphical languages, some capability for automated transformation of designs/specifications to executable forms, and a good capability for building simulations.

- **Specification libraries.**

  A major battle in the specification area is the development of standards for system components. This will be greatly aided by good unified specification languages for both hardware and software interfaces. In ten years we expect standard specifications for often used system components to become commonplace, especially as interface analysis tools develop. It is to be hoped that this approach will resulting in lessening the V & V problem by breaking it down into separate small pieces.

- **Behavioral analysis tools based on formal specifications.**

  Tools for predicting behavior from specifications will become more sophisticated over time. They will evolve slowly, for various methodologies and languages, supporting for example VDM, Anna, and Larch. They will provide much aid in querying specifications, and proving consequences. Their main use will be in analysis of adequacy of specifications by deductive methods. They supply different kinds of techniques from simulators, thereby complementing simulation methods.

- **Self-checking systems, utilizing formal specifications.**

  This is probably the most likely practical approach to constructing secure systems, in which failure may result from a multitude of factors at every level — hardware, compiler, and software errors (including viruses). By ten years, technology of down-loading specification checking onto spare processors should be well-advanced and utilized in production systems. Some mathematical verification will probably be incorporated in runtime checking.

- **Multi-processor simulators.**

  It is to be hoped that there will be significant advances in utilizing concurrency in a system to speed up its simulation on multiple processors. In ten years we may expect quite large numbers of processors to be available for a typical simulator. In this time frame, we ought to expect simulators to be modular, and reconfigurable during simulation to allow for replacement of components by better components. Simulations of some kinds of systems may continue for extensive periods of time, and should be replayable from any historical point.

4. Research projects at Stanford.

The Program Analysis and Verification Group at Stanford has designed formal specification and annotation languages to support Ada software development. These languages are Anna, an Annotation Language for Ada [9], and Task Sequencing Language [7,8]. In addition, the Group has recently developed a high level hardware design language, VAL (VHDL Annotation Language) for the VHSIC program [18,6]. The current research of the Group into advanced system development environments is based on experience with these languages, and tools supporting their application.

- **Anna**

  Anna was first proposed in 1980 [3] and has been under design and development since 1983. Anna is an extension of the Ada language to provide facilities for formally specifying the intended behavior of Ada programs. It was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation could be applied to Ada programs. Anna has already been used by other research groups as a basis for other design and specification languages related to Ada[1] and SDI [15].

1. E.g., the ESPRIT PANDA project, and the Polyanna language at Odyssey Research Associates.

Several tools supporting various applications of Anna to testing and to specification analysis are being implemented, and distributed for experimentation and research [10].

A prototype environment supporting applications of Anna to testing and debugging of Ada programs is currently being distributed worldwide. Methods of applying these tools to V & V activities are being developed and documented. Two near-term goals of this research are:

1. Automated support within Ada environments for testing and debugging utilizing specifications and annotations.
2. **Self-checking Ada systems.**
   Anna package specification analyzers are being constructed. These tools are at a preliminary prototype stage where they can be demonstrated on various package specifications, e.g., packages constituting primary components of a database implemented in Ada.

- **TSL, Task Sequencing Language**

  TSL [7,8] has been designed specifically for specifying concurrent Ada systems. TSL provides facilities for defining abstractly the tasking activity of an Ada system. It is a simple language extension of Ada, containing four principle constructs: *(i)* actions, *(ii)* properties, *(iii)* constraints, and *(iv)* event descriptors. These facilities permit specification of interactions between parallel threads of control (tasks) in an Ada program. Concurrent and distributed programs are specified using behavior patterns expressed in TSL. Patterns can express simple sequences of task interactions, or more complex orderings of interactions — as might be expected in a distributed system.

  TSL specifications are included with the Ada text and are monitored during the computation of the program. Violations of TSL specifications are detected automatically. TSL runtime monitoring techniques are being developed from those used in earlier work to detect classical errors (deadlock and blocking) in Ada programs [14,11,12,13].

  At present a prototype TSL system, including a preprocessor and runtime monitor, is available for distribution. We are currently developing and documenting methods of using TSL as a testing and debugging tool for concurrent Ada programs.

- **VAL, a new high level Hardware Specification Language [18]**

  VAL is a formal language for specifying behavior of hardware designs whose architectures are expressed in VHDL. VAL includes new features that are not in VHDL for expressing: *(i)* timing [6], *(ii)* abstract models of device state, *(iii)* trees of parallel processes, and *(iv)* hierarchical structure. A tool for automatically comparing behavior specified in VAL with a VHDL architecture simulation is being implemented. It thereby provides a capability for automatic comparison of behavior of different levels of a VHDL hierarchical design during simulation.

This previous research on Anna, TSL, and VAL, has demonstrated that formal methods can be applied throughout the systems development process, particularly to Ada-based systems. Systems development activities to which we are currently applying formal methods experimentally include design, specification, simulation, V & V, and self-checking systems. Applications to distributed systems, and to systems containing both hardware and software components, are being researched.

4.1 Implications for SDIO

As we mentioned above, introducing formal methods into the systems development process must be based on sufficiently powerful automated capabilities. There is a great deal of development work involved in taking a prototype tool from the "proof of concept" stage to a production quality environment tool. This step is often overlooked in research programs. It indeed involves research — e.g., in the algorithms and data structures for efficient execution of the tools, modular design structure, user interface facilities, and integration with other environment tools. It is not simply engineering.

We have always emphasized modular design of our support tools for specification analysis, and for annotation checking and debugging. All tools are implemented in Ada. Tool components have modular interfaces, specified in Anna/TSL, through which they interact with other components [10] — this includes, for example, theorem proving components. One of the goals of this emphasis is the eventual re-engineering and integration of this toolset into various fielded Ada environments. Some of the Anna/TSL tools are mature enough to consider doing this.

There are two possible approaches to re-engineering the Anna and TSL toolsets:

1. **development of individual tools for production Ada environments.**

   This is an approach that could be taken with the Anna package specification analyzer, the Anna and TSL checking systems, or the VAL/VHDL checker. Such an effort should lead to a production quality tool within 3 years, and would involve:

   - re-engineering from original Ada source code
   - user interface improvements
   - writing methodology guides/manuals

2. **integration of tool capabilities into standard Ada environment tools.**

   This would be most appropriate for the Anna and TSL checking systems.

   - Compilation of Anna/TSL checking.

     A commercial environment tool such as an Ada compiler would be extended to provide options for compilation of some subset of Anna/TSL annotations, together with suitable diagnostics or additional predefined exceptions.

   - Annotation-based debuggers.

     Checking algorithms for Anna and TSL would be incorporated into a commercial Ada debugger. This would provide an extremely powerful tool for testing and debugging of both sequential and concurrent Ada systems.

SDI Simulation

As a long-term project, it would be advisable for SDIO to support development of a unified systems specification language that provides specification constructs for both hardware and software components. It should include timing constructs for real-time specifications (perhaps along the lines of constructs in VAL/VHDL), but should also be capable of specifying distributed systems in which there is no concept of global time. It should also contain constructs implicit in the current SDI architecture dataflow modeling technique [15]. Development of simulators for the unified SDIO specification language should also be supported.

References

[1] *The Ada Programming Language Reference Manual*. US Department of Defense, US Government Printing Office, 1983. ANSI/MILSTD 1815A Document.

[2] *VHDL Language Reference Manual*. Intermetrics, Inc., Bethesda, Maryland, version 3-5 edition, May 1984.

[3] B. Krieg-Brueckner and D.C. Luckham. Anna: towards a language for annotating Ada programs. *Proceedings of the ACM SIGPLAN Symposium on the Ada Programming Language*, 15(11):128-138, December 1980.

[4] D. Bjorner and C. Jones. *Formal Specification and Software Development*. Prentice-Hall, 1982.

[5] D.C. Luckham. Research on specification languages for real-time systems. July 1988. Proposal submitted to Office of Naval Research.

[6] D.C. Luckham, A. Stanculescu, Y. Huh, and S. Ghosh. The semantics of timing constructs in hardware description languages. In *Proceedings of IEEE International Conference on Computer Design, ICCD'86*, pages 10-14, October 1986. Also Program Analysis and Verification Group Report PAVG-32.

[7] D.C. Luckham, D.P. Helmbold, D.L. Bryan, and M.A. Haberler. Task sequencing language for specifying distributed Ada systems. In *PARLE: Parallel Architectures and Languages Europe*, pages 444-463, Springer-Verlag, 1987. Proceedings on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, June 1987.

[8] D.C. Luckham, D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. Task sequencing language for specifying distributed Ada systems. In *System Development and Ada*, pages 249-305, Springer-Verlag, 1987. Springer-Verlag Lecture Notes in Computer Science, No. 275, Proceedings of the CRAI Consorzio per le Ricerche e le Applicazioni di Informatica Workshop on Software Factores and Ada, Capri, Italy, May, 1986.

[9] D.C. Luckham, F.W. von Henke, B. Krieg-Brueckner, and O. Owe. *ANNA: A Language for Annotating Ada Programs*. Volume 260 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[10] D.C. Luckham, R. Neff, and D.S. Rosenblum. An environment for Ada software development based on formal specification. *Ada Letters*, VII(3):94-106, May,June 1987.

[11] D.P. Helmbold and D.C. Luckham. Debugging Ada tasking programs. In *Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pages 96-110, IEEE, St.

Paul, Minnesota, October 15-18, 1984. Also published, Stanford University Computer Systems Laboratory TR-84-262, July, 1984, Program Analysis and Verification Group Report PAVG-25.

[12] D.P. Helmbold and D.C. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47-57, March 1985. In Proceedings of the IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp. 96-110. IEEE, St. Paul, Minnesota, October 15-18. 1984. Also published as Stanford University CSL TR.84-263, July, 1984.

[13] D.P. Helmbold and D.C. Luckham. Runtime detection and description of deadness errors in Ada tasking. *ACM Ada Letters*, IV(6):60-72, May-June 1985.

[14] D.P. Helmbold and D.C. Luckham. *Runtime Detection and Description of Deadness Errors in Ada Tasking*. CSL Technical Report 83-249, Stanford University, November 1983. Program Analysis and Verification Group Report PAVG-22.

[15] J.L. Linn, C.D. Ardoin, C.J. Linn, S.H. Edwards, M.R. Kappel, and J. Salasin. *Strategic Defense Initiative Architecture Dataflow Modeling Technique*. IDA Paper P-2035, Institute for Defense Analyses, Alexandria, Virginia, 1987. Version 1.5.

[16] J.V. Guttag, J.J. Horning, and J.M. Wing. The larch family of specification languages. *IEEE Software*, September 1985.

[17] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in Five Easy Pieces*. Technical Report 5, Digital Systems Research Center, July 1985.

[18] L.M. Augustin, B.A. Gennart, Y. Huh, D.C. Luckham, and A.G. Stanculescu. Verification of VHDL designs using VAL. In *Proceedings of the International Conference on Computer Aided Design*, pages 418-421, IEEE, Santa Clara, November 1987.

Critical Gaps and Potential Achievements in Software Validation

A Report for the Institute for Defense Analyses

from

Debra J. Richardson
Information and Computer Science
University of California
Irvine, California 92717

1. Critical Gaps

The need to produce highly reliable software systems is becoming more paramount as computers are being introduced into increasingly more critical applications. Strategic Defense System software is of the most risky and complex variety envisioned today and is inherently different from other application areas. First and foremost, no system of comparable size has ever before been developed. Second, the characteristics of the operating environment cannot be known in advance of actual use due to the nature of the application. Third, the system must be reconfigurable due to rapidly changing requirements, some of which will arise dynamically due to countermeasures by the opponents. Fourth, any SDS must perform massively parallel computations distributed on a network of complex system components. Fifth, there will be hard real-time deadlines that must be met by SDS software to ensure the safety of the system and environment. Sixth, the system must be fault-tolerant. These added complexities and the life-critical nature of a SDS make advances in the state-of-the-art of software engineering and, in particular, software reliability ever more crucial.

Although computer engineers are capable of producing computer hardware with ultra-high reliability, such confidence in software engineering has not yet been earned. Current software design techniques cannot guarantee the development of highly reliable software, and formal verification systems are not yet able to deal with software of the size and complexity of today's applications. Neither is today's testing technology capable of assuring much more than intuitive confidence in software reliability. Clearly, current validation technology is not capable of guaranteeing the reliability required by SDS software.

I believe the following critical gaps in the area of software testing and validation must be approached before sufficient confidence in software reliability can be provided: 1) formal software specification techniques must be developed so that software products may be checked for consistency with requirement specifications; 2) formal error detection capabilities must be determined for testing strategies; 3) methods for the integrated application of testing strategies must be developed; 4) an automated support environment must be provided for the use of those strategies; 5) experimental studies to evaluate the practical effectiveness of testing techniques must be performed; 6) more sophisticated techniques must be defined for testing and analysis of concurrent and real-time software; 7) testing techniques must be developed that are applicable in the earlier phases of the software lifecycle; and 8) mature testing technology must be more readily transferred to practicing software testers.

Validation is the process of determining if certain aspects of a software product are consistent with requirements. Without formal specification techniques, those requirements are not amenable to analysis and comparison with the software. This comparison should take place at each stage of the software lifecycle, requiring each new description of the product under development to be consistent with the previous description. Current specification technology is inadequate for such a feedback process.

Although there are a number of interesting testing techniques, very little is known about their effectiveness. The software testing area has suffered from a lack of evaluative work. Many proposed techniques have been justified merely through intuitive arguments, such as the early statement, "one cannot have confidence in a piece of software, if not all of its statements have been executed" (certainly true, but what confidence do we have if all statements are executed). A formalization of the error detection capabilities of the various testing techniques would be much more satisfying and would truly justify the use of these techniques. This requires not only formal models that describe testing techniques, but also classifications and formal descriptions of errors. Moreover, the inherent infeasibility of predicting the actual operating environment of SDS software makes formal error detection a more justifiable approach than any testing approach based on the expected distribution of the input space.

Not only has little research been done on quantifying the effectiveness of individual testing techniques, but neither has much been determined about how they relate to one another. This is particularly problematic for software testing, since it appears that no one technique will be sufficient to achieve the high reliability required by today's life-critical applications. Many testing techniques are geared toward the detection of particular classes of errors (formal error classification and detection capabilities would certify this). Thus, some combination of techniques must be integrated to enable rigorous and comprehensive error detection. Efforts in determining what constitutes a complementary set of techniques must consider trade-offs in terms of cost and rigor. Methods for effectively integrating testing techniques must also be developed.

This leads us to the current lack of an automated support environment for testing. Because of the complexity of some of the testing techniques, testers cannot be expected to perform the level of testing discussed above without such support. This is especially critical for SDS software, which will be on a scale that has never before been installed. Prototype tools that automate individual techniques exist, but little success has been seen in the area of environments to assist in the application of a range of techniques. Most techniques rely on much of the same information garnered from the software by analysis tools (such as control and data flow information), and it is extremely inefficient to redevelop this information for each tool application. Another requirement of such a testing environment is the management of the testing process; test plans and histories (along with attributes affiliating test cases with software products) must be retained throughout the development and maintenance processes. This retention of information on previous testing phases can be extremely helpful in regression testing after modification of the software. With an ever changing system such as SDS, efficient re-validation is an extremely important process.

The implementation of a testing environment will also allow firsthand experience with testing techniques and experimental studies. Due also to the complexity of techniques, such an environment is the only feasible way to experiment with alternative approaches. Theoretical studies into the error detection capabilities of approaches can provide considerable insight into their relationships but not empirical evidence of their relative value. Two techniques of similar error detection capabilities may make very different demands on the tester, perhaps making one technique much less cumbersome to employ than another. We are concerned not only with the practical error detection capabilities of approaches, but also on the run-time requirements of the technique. Experimental studies are particularly important in testing research since worst case analysis often gives very pessimistic upper bounds that do not satisfactorily reflect typical performance.

Most testing research has targeted sequential programs, although the most critical software applications these days are concurrent and/or real-time software. The parallelism in SDS software will be massive and the real-time constraints absolute; neither can be ignored. The sequential testing techniques can be applied for testing the functional aspects of regions of concurrent programs and a few approaches to testing the interactions between concurrent processes have been proposed, but little has been done on integrating these two aspects of the validation problem. Any such integration will most likely rely upon a scheduler to select potential interactions. This is a particularly difficult problem for SDS software as it is extremely difficult to predict the likely interactions between the sub-systems/processes. Another problem in this arena arises due to the common practice of developing real-time systems on a host machine that differs (potentially drastically) from the target machine. The advantages of this approach are that the development environment can provide the development (and testing) environment, but the testing performed on the host machine may not be valid on the target machine.

It has been repeatedly shown that it is more effective to detect and correct an error as early in the lifecycle as possible. Testing techniques that are applicable throughout the full software lifecycle must be developed. This may involve the application of existing techniques to pre-implementation descriptions or the development of new techniques that are geared toward the use of more abstract descriptions of software.

Finally, the state-of-the-art in software testing is less mature than that of software applications, while the state-of-the-practice in testing is falling notably further and further behind. Another major gap that must be bridged is the lack of transfer of the testing technology that is maturing in the research community.

2. Potential Achievements

In the next few years, I believe that some of the shortcomings in testing technology can be overcome. Work is progressing in analyzing the formal error detection capabilities of existing testing techniques [Rich86a]. Several prototype tools that support individual techniques already exist [Clar76,DeMi88,Fran85,Luck86,Oste76,Wu87], and the development of a testing environment that will integrate complementary testing techniques is underway [Clar88].

Previous work with respect to validation techniques that are applicable to concurrent and real-time software [Avru86,Brin85,Dill88,Luck87,Tayl85,Youn86] appears promising, although most of these techniques do not explicitly address the testing of concurrent software. In the long-term, however, these techniques are coming [Tayl86]. In addition, researchers have contemplated testing techniques that are applicable in the pre-implementation phases of the software lifecycle [Luck86,Rich85]. As specification and design languages become more standard, these techniques will be advanced. Large-scale experimental studies can also be achieved in the long-term by continuing current efforts [Basi85,Basi88,Selb87].

All of this work, especially the urgently needed tool development efforts, will require funding far beyond the levels provided to the software validation community today. The validation field has made relatively slow progress to date, primarily due to the size of the community and the lack of resources.

Lastly, it is not clear that even monumental advances in the state-of-the-art of software validation will provide the capabilities required by SDS software due to the intense safety-critical nature of such a system. This, however, does not constitute a reason to stop pursuing or supporting validation research.

3. Current Research

Dr. Richardson's research has been directed toward program testing methods and their ability to detect potential errors. Although there are a number of interesting testing techniques whose use would improve the reliability of software, the testing of computer programs is typically done in a haphazard manner. Insufficient testing can be extremely costly, especially with today's prevalence of life-critical software. The major reason testing is neglected is that the capabilities of existing techniques are not well understood and their application is cumbersome at best. What is needed is an automated testing environment, composed of an integrated and well understood set of testing techniques, that will provide considerable guidance in the testing process.

To this end, Dr. Richardson[1] is performing research in four general areas: 1) the development of techniques for evaluating the effectiveness of existing testing methods, 2) the design of new and more powerful testing technology, 3) the investigation of ways to integrate testing techniques, and 4) the construction of tools that automate testing techniques within a software development environment. This research is highly synergistic and cyclic: theoretical evaluation naturally leads to recognized needs for new technology; once designed, new techniques must be evaluated; evaluation helps to select techniques that should inhabit a testing environment, which must then be implemented; and implemented techniques must be empirically evaluated. This synergism and incremental development provides an overall project that will substantially affect the ways in which we test software.

3.1 Evaluation and Design of Testing Techniques

We have undertaken an evaluation of a variety of testing techniques with an eye toward two goals: 1) the discovery of which techniques are the most effective, and 2) a better understanding of the types of errors that each technique is particularly adept at detecting. Throughout the evaluation process, we are keeping in view the final goal — the development of a testing environment — by considering how techniques might be improved, what additional techniques are needed, and how complementary techniques can be combined and implemented.

One of the difficulties in carrying out this investigation is that most existing techniques use different underlying models. Thus, one of our goals is to formulate a consistent model for software testing that is expressive enough that most testing techniques can be mapped onto it.

For the most part, testing techniques are directed either at the problem of determining the paths, the particular sequences of statements, that must be tested or at the problem of selecting test data for the selected paths. For the path selection problem, techniques based on control flow and data flow have

1. along with colleagues at the University of Massachusetts at Amherst

been proposed. For the test data selection problem, techniques can be further classified as error-based, geared to reveal errors in the execution of a particular path, or fault-based, directed toward the detection of faults in a particular statement. We began our evaluation of testing techniques by considering path selection and test data selection independently, although we have come to realize that the standard separation is not appropriate.

3.1.1 Path Selection

In our initial work on path selection, we defined a model for describing path selection and compared three families of data flow path selection techniques [Clar85a]. These criteria are concerned with selecting paths that cover particular uses of defined variables. We began with this group of criteria since they appear most promising than the approaches that use an assortment of control flow information. We formulated a uniform model for data flow and defined each of the data flow criteria in terms of that model. This effort alone uncovered ambiguities and discrepancies in some of the criteria.

Using the formal definitions of the criteria, we analyzed the path coverage of each criterion and developed a subsumption hierarchy that demonstrated how these criteria relate to each other [Clar85a,Clar86]. This analysis has shown that the most comprehensive of the three families of criteria are incomparable with each other as a consequence of their different foci.

We would like to continue this investigation by evaluating the coverage of path selection criteria that are based on other sorts of program attributes. To this end, we have been considering a more general model, based on information flow, which attempts to capture control flow, data flow, and implicit data flow relationships.

The graph-theoretic analysis of the path coverage of various criteria has provided interesting insights into the strengths and weaknesses of these different techniques. It is only a first step, however, in understanding the path selection problem. Further work must compare the costs of using different criteria, considering both the cost associated with applying each criterion as well as the cost of actually testing the selected paths. The graph-theoretic nature of the model should facilitate this comparison. Moreover, this aspect of the model will also enable the modification of the criteria so that they can realistically deal with common program phenomena such as infeasible paths.

Another area of crucial research is the determination of the true nature of the contribution of the different path selection criteria. It is important to understand the effects of the differences between the criteria in terms of error detection and not only in terms of path coverage. We intend to use the data/information flow model in formulating classes of errors upon which to evaluate the different approaches to path selection. We hope to be able to state the conditions under which a particular path selection criterion is capable of guaranteeing the detection of a particular class of errors. This will provide even more insight into the relative strengths of the criteria.

3.1.2 Test Data Selection

We began informally evaluating test data selection techniques in our work on symbolic evaluation. Testing systems based on symbolic evaluation typically performed only naive test data selection for the paths that were chosen by solving the path domain. It seemed reasonable to devise techniques that more fully use the information derived by symbolic evaluation [Clar85b,Rich85]. This led to the development and evaluation of error-based test data selection techniques. These error-based techniques had in the past been rather ad hoc applications of long-touted heuristics. Using the symbolic representation of a path, however, allowed the more formal definition of error-based techniques [Clar83]. Such techniques are aimed at selecting test data to illuminate specific types of errors, categorized as domain errors or computation errors.

While looking at symbolic evaluation and the associated error-based techniques, we became concerned that most testing methods, especially automated ones, are based solely on analysis of the program implementation and ignore its specification. By so doing, such methods test what the program does rather than what it is supposed to do. The *Partition Analysis Method* [Rich81,Rich82] was conceived of in search of a better approach to testing. Partition Analysis partitions the set of input data into meaningful subdomains by applying symbolic evaluation techniques to both the specification and the implementation of a program. This partition forms the basis for the selection of test data and the verification of consistency between the two descriptions of the program.

As a program validation method, Partition Analysis has several important distinctions. First, it integrates several complementary testing techniques. Second, the selected test data appropriately characterize both the program's intended behavior (as described by the specification) and its actual behavior (as characterized by the structure of its implementation). Third, Partition Analysis can be used throughout much of the software lifecycle since it is applicable to a number of different types of specification languages, including both procedural and nonprocedural languages. Fourth, the testing and verification processes are unified within Partition Analysis so as to complement and enhance each other.

To further our evaluation of test data selection techniques, we have been trying to formulate a model that would be powerful enough to support the comparison of test data selection techniques. Because of the wide diversity of approaches, this has proven to be more difficult than the formulation of the graph-theoretic model developed for path selection. We are also striving to keep the model for test data selection consistent with the model for path selection. Recently, we came up with a model that appears to have the power and flexibility that is needed. This model, called RELAY (because it views the process of error detection as analogous to a relay race), has been used to define several test data selection criteria [Rich86a] and to formulate a model for error detection [Rich86b,Rich88].

RELAY models the process of error detection by defining *revealing conditions* on a set of test data that guarantee that a fault in the source code is actually revealed as an error in execution. An error is revealed if the fault *originates* an error during execution and that error *transfers* through all computations and data flow until it is revealed in the output. RELAY has three principal uses: 1) as a method for evaluating other testing techniques; 2) as a test data selection technique; and 3) as a test data measurement technique.

For the purposes of evaluation, RELAY provides a sound method for analyzing the error detection capabilities of other testing criteria [Rich86a]; it is particularly suited to those that are fault-based. We have analyzed three fault-based criteria, each of which attempt to detect faults in the six fault classes: Budd's *Estimate*, Howden's *Weak Mutation Testing*, and Foster's *Error-Sensitive Test Case Analysis*. This analysis demonstrated that none of these criteria guarantees the detection of these types of faults and points out two common weaknesses. First, the criteria do not thoroughly consider the potential unsatisfiability of their rules; each criterion includes rules that are sufficient to reveal errors for some fault classes, yet when such rules are unsatisfiable, many errors may remain undetected. Second, the criteria fail to integrate their rules; although a criterion may cause an expression to take on an erroneous value, there is no effort made to guarantee that the enclosing expressions evaluate incorrectly.

The RELAY model of error detection also provides a fault-based criterion for test data selection [Rich86b,Rich88]. RELAY is applied by choosing a fault classification and determining the origination conditions and transfer conditions for each class of faults. These conditions are then evaluated for the program being tested to provide the revealing conditions. These conditions can be used to select test data or as a metric for test data selected by another technique. Execution of the program for test data that satisfy these conditions guarantees the detection of errors that result from any fault in the chosen classes. RELAY overcomes the flaws common to other fault-based techniques because the test data selected by RELAY satisfies precise origination conditions that are coupled with transfer conditions. Moreover, these conditions are necessary and sufficient for revealing an error.

We continue the development and evaluation of the RELAY model. One major area of research is the application of the model to a more extensive and diverse classification of faults. We intend to investigate the use of RELAY throughout the software lifecycle, which we believe will not only provide a means of defining more abstract fault classes but also provide a means of analysis in the earlier phases of the lifecycle. We are enhancing the capabilities of RELAY with regard to data flow transfer, thus enabling the processing of loops in an efficient manner and accounting for the transfer of multiple errors. Throughout, we will continue to evaluate the error detection capabilities of the RELAY model both analytically and empirically.

3.2 Integration and Implementation

It is unlikely that any single testing technique will ever be found that is sufficient to test all programs for all types of faults and errors. Instead, a set of complementary techniques must be integrated to be capable of demonstrating a higher degree of reliability. The long-term goal of this research is the design of a testing strategy that integrates complementary techniques and the construction of a testing environment to demonstrate the feasibility of and permit the evaluation of this testing strategy.

We are convinced that to achieve rigorous error detection through an automated testing system, a number of techniques must be carefully integrated. In terms of path selection, it would be relatively easy to propose a new criterion that combines all of the useful criteria. It would not be clear, however, what would be the benefits of such a union. The same could be said for test data selection. As previously mentioned, a primary concern of our work is the determination of what constitutes a complementary set of techniques that provides rigorous error detection capabilities. The data/information flow and RELAY models provide us with a foundation for comparing the error detection capabilities of the different techniques. We would like to characterize the trade-offs in selecting this set in terms of cost and rigor.

For the most part, testing research has separated concerns by considering either path selection or test data selection. Our experience with RELAY, however, has shown that this usual separation just doesn't work. RELAY has shown that test data selection must be performed in conjunction with path selection so that test data is selected that not only originates an error but also transfers that error to affect the output. Thus, we must carefully consider how path and test data selection should be integrated. The investigation into possible integrated approaches to software testing raises a number of interesting research questions. Many testing criteria, including the data flow path selection and the fault-based test data selection, can be realized as measurement tools that provide a metric on user-selected data/paths or as fully automated selection tools. The difference in cost between these two approaches and alternatives among these two extremes must be considered as well as the ways in which these two approaches might be achieved together.

In parallel with the other efforts, we are initiating the construction of a prototype testing environment, called TEAM (Testing, Evaluation, and Analysis Medley) [Clar88]. The TEAM environment strives to support: 1) integration of diverse testing and analysis tools, 2) extensibility of that tool set so that new tools can easily be added, 3) experimentation with different approaches to software reliability, and 4) full software lifecycle testing and analysis.

Although at this time we are not certain what techniques must populate the TEAM environment, our evaluation work has clearly indicated certain basic capabilities must be provided. These basic analysis components provide the building blocks for the creation and integration of more sophisticated testing techniques within the TEAM environment. We have also designed the components as generic capabilities whenever possible. Generic components provide capabilities that can be instantiated to meet the needs of different testing tools in the environment. To support and further new testing techniques, particularly those that address the pre-implementation phases of the lifecycle, it is important that the TEAM environment support an assortment of languages, including specification, design, and coding languages. To accomplish this, we have designed the system to be relatively language independent.

As a part of the TEAM project, a prototype testing tool based on the RELAY model will be developed so as to demonstrate the potential of RELAY as a testing method and its integration with other testing techniques. Other testing tools will be included as well to demonstrate the prospect of technique integration.

3.3 Conclusion

We are currently pursuing all four research areas in parallel. Our research on test data selection provides insight about path selection and vice versa. The initial prototype of the TEAM environment will contain basic data flow and symbolic evaluation capabilities and should be running by the end of the year. At that time we can begin to get experience with the system and start the design of the more sophisticated path selection and test data selection techniques that, as indicated by our studies, most effectively provide rigorous error detection capabilities.

Based on the preliminary evaluation of testing techniques and the initial development of the testing system, our goal of developing an extensive, powerful testing environment seems achievable and promises to substantially enhance the reliability of software tested with such a system.

References

[Avru86] Avrunin, G.S., L.K. Dillon, J.C. Wileden, and W.E. Riddle. "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems." *IEEE: Transactions on Software Engineering*, 12/2 (Feb 1986):278-292.

[Basi85] Basili, V.R., and R.W. Selby. 1985. "Calculation and Use of an Environment's Characteristic Software Metric Set." In *Proceedings 8th International Conference on Software Engineering*, London, England, 386-391. Washington, DC: IEEE Computer Society Press.

[Basi88] Basili, V.R., and H.D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environments."

[Brin85] Brindle, A.F., R.N. Taylor, and D.F. Martin. September 1985. *A Debugger for Ada Tasking*. El Segundo, CA: The Aerospace Corp. ATR-85(8033)-1. Also published in *IEEE: Transactions on Software Engineering*, 15/3 (Mar 1989):293-304.

[Clar76] Clarke, L.A. "A System to Generate Test Data and Symbolically Execute Programs." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):215-222.

[Clar83] Clarke, L.A., and D.J. Richardson. 1983. "A Rigorous Approach to Error-Sensitive Testing." In *Proceedings IEEE 16th Hawaii International Conference on System Sciences*, January, Honolulu, HA.

[Clar85a] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1985. "A Comparison of Data Flow Path Selection Criteria." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 244-251. Washington, DC: IEEE Computer Society Press.

[Clar85b] Clarke, L.A., and D.J. Richardson. "Applications of Symbolic Evaluation." *Journal of Systems and Software*, 5/1 (1985):15-35.

[Clar86] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1986. "An Investigation of Data Flow Path Selection Criteria." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 23-32. Washington, DC: IEEE Computer Society Press.

[Clar88] Clarke, L.A., D.J. Richardson and S.J. Zeil. "Team: A Support Environment for Testing, Evaluation, and Analysis." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 121-129.

[DeMi88] DeMillo, R.A., D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. 1988. "An Extended Overview of the Mothra Software Testing Environment." In *Proceedings 2nd Workshop on Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 142-151. Washington, DC: IEEE Computer Society Press.

[Dill88] Dillon, L.K., R.A. Kemmerer, and L.J. Harrison. 1988. "An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of Ada Tasking Programs." In *Proceedings 2nd Workshop on Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 114-121. Washington, DC: IEEE Computer Society Press.

[Fran85] Frankl, P.G., S.N. Weiss, and E.J. Weyuker. 1985. "ASSET: A System To Select and Evaluate Tests." In *Proceedings IEEE Conference on Software Tools*, April, New York, 72-79.

[Luck86] Luckham, D.C., R. Neff, and D. Rosenblum. August 1986. *An Environment for Ada Software Development Based on Formal Specification*. Stanford University. Technical Report CSL-TR-86-305. Also published in *ACM: Ada Letters*, VII/3 (May-June 1987):94-106.

[Luck87] Luckham, D.C., D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. July 1987. "Task Sequencing Language for Specifying Distributed Ada Systems." In *Proceedings of CRAI Workshop on Software Factories and Ada*, Capri, Italy, A.N. Habermann and U. Montanari (eds). Lecture Notes on Computer Science, 275. Springer-Verlag, 249-305. Also published as Stanford University Technical Report CSL-TR-87-334.

[Oste76] Osterweil, L.J., and L.D. Fosdick. "DAVE – A Validation Error Detection and Documentation System for Fortran Programs." *Software Practice and Experience*, 6/4 (Oct-Dec 1976):473-486.

[Rich81] Richardson, D.J., and L.A. Clarke. 1981. "A Partition Analysis Method to Increase Program Reliability." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 244-253. Washington, DC: IEEE Computer Society Press.

[Rich82] Richardson, D.J., and L.A. Clarke. 1982. "On the Effectiveness of the Partition Analysis Method." In *Proceedings 6th International Computer Software and Applications Conference*, March 9-12, San Diego, CA, 529-538. Los Angeles, CA: IEEE Computer Society.

[Rich85] Richardson, D.J., and L.A. Clarke. 1985. "Testing Techniques Based on Symbolic Evaluation." In *Software: Requirements, Specification, and Testing*, T. Anderson (ed.), 93-110. Blackwell Scientific Publications.

[Rich86a] Richardson, D.J., and M.C. Thompson. December 1986. *An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection*. University of Massachusetts. Technical Report 86-65.

[Rich86b] Richardson, D.J., and M.C. Thompson. December 1986. *A New Model for Error Detection*. University of Massachusetts. COINS Technical Report 86-64.

[Rich88] Richardson, D.J., and M.C. Thompson. 1988. "The RELAY Model of Error Detection and Its Application." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 223-230. Washington, DC: IEEE Computer Society Press.

[Selb87] Selby, R.W., V.R. Basili, and F.T. Baker. "CLEANROOM Software Development: An Empirical Evaluation." *IEEE: Transactions on Software Engineering*, 13/9 (Sep 1987):1027-1037.

[Tayl85] Taylor, R.N. June 1985. "Debugging Real-Time Software in a Host-Target Environment." In *Proceedings 2nd International Conference on Software Engineering*, October, San Francisco, CA, 194-201. Washington, DC: IEEE Computer Society Press.

[Tayl86] Taylor, R.N., and C.D. Kelly. 1986. "Structural Testing of Concurrent Programs." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 164-169. Washington, DC: IEEE Computer Society Press.

[Wu87] Wu, D., I.J. Riddell, M.A. Hennell, and D. Hedley. April 1987. *The Minimum Set of Test Data on Syntax Directed Mutation of Boolean Expression*. University of Liverpool.

[Youn86] Young, M., and R.N. Taylor. 1986. "Combining Static Concurrency Analysis with Symbolic Execution." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 170-178. Washington, DC: IEEE Computer Society Press.

SDS Software and Analysis Research

Richard N. Taylor

Department of Information and Computer Science
University of California, Irvine 92717

1. Relevant Properties of the Application Areas

The capabilities of current testing and evaluation (hereinafter called "analysis") technology vary with properties of the application area. The pertinent properties for this discussion are that the application software is:

- large (100,000+ lines of higher-level language source code),
- distributed (multiple, separate processing elements communicate and cooperate in the solution),
- real-time (subject to stringent requirements on the time allowed for generation of required outputs).

SDS software will exhibit several additional properties that have consequences with respect to the analysis activity.

- very large size (1-10 million lines of HLL)
- numeric (significant portions of the application involve floating-point calculation)
- fault-tolerant (the system must be built to accommodate operational loss of processing elements, and possibly software faults)
- reconfigurable (system layout and communication patterns may be routinely changed)
- requirements not subject to complete, consistent, formal specification (e.g. characteristics of input data, such as number of entities)
- rapidly changing requirements (The very size of the application ensures that the requirements will change frequently. Moreover the nature of the application demands frequent change, in the endless cycle of measure, counter measure, and counter-counter measure, ...)
- no comparable systems (no system has been built which is comparable in terms of size, complexity, requirements characteristics, need for trustworthiness, etc.)
- inability to test system in the field (the nature of the application is such that no serious field test can be attempted)
- limits on simulation abilities (Useful simulation exercises demand models of the environment that are faithful to reality in all relevant aspects. Similarly the system (code and hardware) must be faithfully modeled. Yet the application is such that we cannot be sure in advance what all the relevant features of the environment are. And the only adequate model of the system is the system itself.)

- extreme real-time performance requirements (combined with lack of time for human involvement)

2. Application Area Requirements

The key demand of the application, of course, is trustworthiness. We must be assured that the system will not fail catastrophically the first, and presumably only, time it is used. The risk of catastrophic failure on less demanding large, real-time, distributed systems, such as flight control systems, is acceptable – there are test pilots and test aircraft. The same circumstances do not hold for SDS.

Trustworthiness must also be maintained. As the environment changes, as the hardware changes, as the software requirements change, the system changes. Yet each system must continuously be trustworthy. This means rapid re-development and rapid re-certification.

Current analysis technology is clearly incapable of providing the required assurances for SDS software. Indeed, it is currently incapable of providing many critical assurances for large systems, distributed systems, or concurrent/real-time systems. Large, distributed, real-time systems are being built, however, and they all experience failures. It is the purpose of this document to indicate what the major problems are with respect to analysis and what can be done to improve the situation.

3. Interaction of Analysis with Development and Maintenance

Though it may be obvious it bears repeating that effective analysis (and that over the life of a project) is closely tied to development and maintenance procedures. Analysis is the checking of consistency between a software object and its requirements. Unless the requirements exist in a form amenable to objective comparison then analysis is not possible. Effective re-analysis of a changed system demands that intermediate software objects, their relations, and attributes must be retained. Otherwise analysis would have to proceed from scratch and would likely be unaffordable, or unacceptable in terms of schedule.

Once management concerns are introduced, then it is also clear that analysis must be done during development, showing consistency of intermediate artifacts with previously developed requirements. If this is not done then the cost and schedule impact of tail-end analysis, with consequent redevelopment, would be such as to guarantee that the system would never be certified and never fielded.

Not only must analysis be done through the entire software process, it must be done for each level of the application and for each supporting technology. Levels of the system are such things as battle management software, communications support layer, operating systems layer, micro-code layer, and hardware. Supporting technologies are translators that enable applications at one layer to be implemented at the next layer down – compilers are one example, design refinement systems are another.

4. Critical Gaps

- Analysis technology does not stand alone. Analysis is related to programming language used, system model, design method, requirements specification, safety requirements, language processing tools, hardware, the assurances required, etc. Progress is required in each component technology area. But progress achieved in one area, in isolation from the others, is very unlikely to yield the improvements required. The interactions between the various areas are complex. Achieving the necessary improvements in each component technology while simultaneously ensuring that the approaches taken in separate technologies are not only compatible, but complimentary, is a daunting task. We do not know how to adjust approaches in the component technologies such that compatibility between approaches is assured.

- Formal specifications of the task to be addressed by software are required. All analysis work is predicated on the existence of specifications against which consistency is checked. If specifications are informal, incomplete, or inconsistent no meaningful assurances can be provided. We do not currently have adequate specification techniques available.
- Not only must software be validated against specifications, but the specifications must be validated against the problem. Rigorous techniques can be brought to bear to compare software, a formal object, with formal specifications. But formal specifications are only a reflection of the real "problem". We do not know how to develop confidence that formal specifications capture all the necessary components of the problem.

  (One must also be assured that the *models* of the physical world, on which the software is based, are adequate representations of the physical world, in which the "problem" occurs.)
- Conventional mathematics is not an accurate description of the calculations performed by computers. Well-known properties of computer calculations, such as round-off, truncation, and overflow, pose serious problems for numerical analysis. That is, it is difficult to develop adequate mathematical models of the physical world (so that software can be written) that are consistent with the properties of computer computation. What is worse, verification of numeric algorithms must not involve the use of typical mathematical properties such as associativity and distributivity. We do not know how to formally verify algorithms involving floating-point arithmetic.
- Supporting analysis of large, distributed, real-time systems requires an extensive software development environment. Analysis involves the application of tools embodying techniques, the management of large amounts of data and partial results, methods for directing analysis based on incremental information, and mechanisms for effectively presenting, justifying, and explaining results to humans. We do not know how to build such an environment on the required scale.
- Analysis demonstrates the presence or absence of specified properties of software. Failures in fielded systems sometimes result from the presence of unexpected or previously unidentified critical properties. We do not know how to determine what all the critical properties of a system are.
- No single analysis technique can pretend to provide all the assurances required – multiple techniques must be employed. Current analysis techniques operate on the basis of various assumptions; these assumptions are sometimes inconsistent. We do not know what a workable set of common assumptions is.
- Reliability assessment techniques make assumptions about input data distributions and properties of software errors. We do not know if these assumptions are admissible.
- Two major tools for dealing with the problems posed by large and complex systems are modularity and abstraction. Where analysis of a large system as a monolith may be intractable, analysis may be feasible if the system can be broken into a hierarchy of modules such that each module may be analyzed separately, summarized as a useful abstraction, and then used in the analysis of higher level modules. We do not know what are all the appropriate testing and analysis abstractions for *large, distributed and real-time* systems, or how to combine abstractions in the analysis of higher-level modules.

- Current techniques for the verification of critical real-time performance properties are based on worst-case analyses in the presence of a fixed schedule of static program tasks on a single processor. We do not know how to deal with situations where the number of tasks is dynamic, the structure of the problem is flexible (due to fault-tolerance reorganizations of both software and hardware), and there are multiple processors available to work on a task.
- Current analysis technologies have been applied only to small problems. We do not know how the techniques will scale, or even what the critical scaling factors are.
- Analysis is a costly and time-consuming task, in terms of both human and machine resources necessary. Analysis is done with respect to a specification. When the specification changes or the software changes, the system must be re-analyzed. We do not have adequate techniques for determining what parts of a system must/need not be re-analyzed.
- Dynamic creation/termination of objects and concurrent processes is a common property of large distributed systems. We do not have adequate techniques for analyzing such systems.

There are also some relevant social factors.

- Analysis, as a field, has made slow progress. There are relatively few researchers in the area. There is no reason to expect the pace of progress to dramatically increase.
- Due to limited resources and the difficulty of conducting experiments, there has been little empirical evaluation of analysis techniques and very little repetition of experiments. Thus what confidence we currently have in techniques is based largely on paper arguments. It seems that unwarranted confidence is placed on the technical reviewing process.

5. Expectations and Beliefs

Having indicated what the critical gaps in analysis technology are, we now indicate what we feel can be changed, how fast it can be changed, and what course of action will lead to the most effective change. These beliefs and expectations can be evaluated by looking at recent progress in the field and at typical technology transition times.

The key observations from the preceding list of critical technology gaps are as follows:

- fundamental new technology must be developed
- various technologies must be integrated
- adequate support infrastructure must be built
- empirical studies of technologies must be performed

We must furthermore keep in mind that technology transition – from concept formation to industrial use – typically takes ten years.

My guess, therefore, is that it will take at least ten years to develop the basic technology and ten more for the transition. Thus it will be at least twenty years before we can begin to apply the necessary analysis technology to SDS software.

Continuing in this pessimistic line, "at least ten years" for the technology development means that the requisite technology can probably be developed if significant new research and development activity occurs in individual technology areas and in integrative technology. New activity requires

- better organization and cooperation among researchers in the pertinent fields
- additional researchers in the field
- better review and evaluation of research
- adequate mechanisms to support technology development *and review* (empirical evaluation through funding for programmers and through provision of adequate hardware support).[1]

I believe that the highest payoff areas are:

- support infrastructure (environments) – because the entire software process must be supported effectively to enable effective analysis. Moreover rapid reconfiguration, change, and recertification demand incremental capabilities,
- abstraction mechanisms supporting development and analysis of large systems – because apart from adequate mechanisms for parceling the work involved there is no hope for either system construction or analysis,
- models supporting multiple, integrated analysis techniques – because no single technique will be adequate, multiple techniques must be used, and the semantics of joint use must be fully understood.

6. Why the Research Must Be Done

The preceding list of issues with regard to analysis of SDS software is not encouraging. Yet those issues should not be taken as an argument for not pursuing research in the analysis area. On the contrary, large, distributed, real-time systems are being built now – some of which are destined for military application. We must develop the capability to ensure that such systems are trustworthy in their application area. We do not currently have *that* capability. It is because the demands on such systems are less stringent than on SDS that gives hope that the requisite software analysis technology can be developed.

7. Research in Support Infrastructure (Taylor)

Effectively supporting analysis of systems and software requires a extensible, integrated software environment. Such as environment must contain facilities for directly supporting software analysis processes (as well as associated development and maintenance processes), a persistent typed object management system, and facilities for enabling effective communication between humans and computers. The processes involved in analysis should be pro-active, can be complex, involve interaction between many tools, and make significant demands on human users. The multiplicity of tools involved in

1. "Magic solutions" – the cure-all technologies that will rescue a project – are commonly touted, and gotten away with for a time, simply because we do not currently have adequate infrastructure, technology, personnel, inclination, or habit to scrutinize proffered silver bullets and perform repeat experiments.

an analysis process communicate through a variety of software objects whose attributes and interrelationships are complex yet essential aspects of the information. Effective use of the tools demands that information be presented to users in appropriate form (graphical or textual), the personnel involved can interact with the environment in consistent, effective ways, and the user interface can evolve rapidly to present new views of objects, to make use of new hardware, and to support new tools.

In conjunction with our colleagues in the Arcadia Consortium, we are involved in creating such an environment. The first prototype, called Arcadia-1, will demonstrate the key technologies and will be available in the fourth quarter of 1990. As a research prototype it will not be production quality, but will be scalable. Subsequent Arcadia environment prototypes will appear in the following years, each with additional functionality.

8. Research in Analysis Technology (Taylor)

8.1 Hybrid Analysis Techniques

Every software analysis technique has some critical limitations. Fortunately the limitations of one technique can often be addressed by also applying another. Doing this, however, requires that the interactions between the techniques are fully understood. Deeper examination often reveals new ways of looking at the analysis techniques and new ways for constructing tools in support of the analyses. Currently, in an activity being led by Michal Young, we are looking at *state-space exploration* as a framework within which to understand the interaction of various tools. Some techniques sample the state-space, others fold it, so as to make tractable the exploration of an intractably large space. Some foldings are particularly interesting as they are error-preserving [Youn88]. While this is initially a theoretical exercise, it has practical applicability, for example, in the combination of static concurrency analysis and symbolic execution [Youn86].

A more general view than this is being taken in our investigation of the nature of software quality assurance arguments. The thesis of the work is that any argument for software quality must be seen in terms of three coordinates: the model of computation with respect to which the argument is made (e.g. ideal mathematics, emulation of a target machine, assumed correctness of some virtual machine), the information on which the arguments are based (e.g. specifications, code), and the properties that are demonstrated with the argument (e.g. functional correctness, performance, safety, fault-tolerance). The purpose of the investigation is to help us know what we know – and what we don't – after software quality assurance is "done".

Initial investigations in the hybrid analysis area are expected to be complete within a year. Moving from these initial theoretical investigations to products will take several years, however. The investigation of SQA arguments is just beginning. Substantive progress is dependent on identification of appropriate personnel to further the work. No "product" is anticipated from this work; rather some key insights useful in evaluating the benefits achieved by application of other products is expected.

8.2 Testing Concurrent Software

Testing of concurrent software poses new challenges in addition to all those of sequential software. The focus of this research activity is to create a methodology for testing concurrent programs. The first steps being taken are designed to develop notions of test *coverage*. While coverage metrics for sequential software are of limited benefit, there is to date nothing of comparable value for concurrent systems [Tayl86]. Supportive technologies for developing test data, controlling tests, and monitoring concurrent program activity are likewise being developed. Further activities will quickly move to more substantive issues of comparing program behavior (via tests) with specifications of the desired concurrent behavior.

Substantive progress in this area is dependent upon adequate infrastructure on which to base the techniques and adequate development of supportive analysis technology, most notably static concurrency analysis. This infrastructure and support technology should be ready in two years. Development of an initial methodology and supporting tool set could take place in an additional 2-3 years.

8.3 Debugging Concurrent Software in a Host/Target Context

Embedded systems software development does not typically take place on the embedded processor. Instead one attempts to perform as much development and analysis on a host machine, then cut over to the target when finally necessary. A problem arises when a fault is observed on the target machine and that problem cannot be duplicated in the host environment. The problem is common and serious in flight control systems and shipboard command and control systems.

Our strategy for approaching this issue was outlined in a paper in 1985 [Tayl85]. The basic ideas are to support a hierarchy of test situations (ranging from purely on-host testing to purely on-target testing) and to perform all debugging (and associated analysis) on the host through, if necessary, reconstruction of a failed target execution.

A wide variety of techniques and tools are involved in the solution, potentially including a specialized interpreter, a static concurrency analyzer, a symbolic executor, assertion checkers, and so on.

The approach needs to be refined and applied in experimental situations to determine its viability. This has not yet happened for several reasons: lack of an adequate infrastructure on which to base the multiple tools/techniques involved, immaturity of the component technologies, lack of an adequate hardware base for the host/target connections and monitors, and lack of adequate personnel.

Our current activities are focused on developing the environment infrastructure necessary and developing the component technologies. As these two activities mature we will be ready to move ahead with the integration and experimental aspects of the work. These two base activities should reach adequate maturity within two or three years. At that point ramped up activity would be appropriate as well as necessary.

8.4 Development Strategies for Concurrent Software

The focus of this activity is to develop design strategies for concurrent software systems that make analysis more effective. The approach is to examine various techniques for doing various kinds of analyses, note the features of programs that make application of the technique difficult or infeasible, then synthesize a design strategy yielding software not exhibiting those properties. The belief is that if a program is difficult to analyze it is difficult to understand.

This activity is just getting underway -- the need is to identify appropriate investigators. Initial substantive results could be expected in 3-4 years.

References

[Tayl85] Taylor, R.N. June 1985. "Debugging Real-Time Software in a Host-Target Environment." In *Proceedings 8th International Conference on Software Engineering*, October, San Francisco, CA, 194-201. Washington, DC: IEEE Computer Society Press.

[Tayl86] Taylor, R.N. and C.D. Kelly. 1986. "Structural Testing of Concurrent Programs." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 164-169. Washington, DC: IEEE Computer Society Press.

[Youn86] Young, M. and R.N. Taylor. 1986. "Combining Static Concurrency Analysis with Symbolic Execution." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 170-178. Washington, DC: IEEE Computer Society Press.

[Youn88] Young, M., R.N. Taylor, D.B. Troup, and C.D. Kelly. 1988. "Design Principles Behind Chiron: A UIMS for Software Environments." In *Proceedings 10th International Conference on Software Engineering*, April 1988, Singapore, 367-376. Washington, DC: IEEE Computer Society Press.

Testing and Evaluation of SDS Software through Validation Techniques

Lee J. White
CASE Institute of Technology
CASE Western Reserve University
Cleveland, OH 44106

Critical Gaps in Validation Techniques for Testing and Evaluation of SDS Software

The IDA Memorandum Report P-2132 [8] has identified numerous critical gaps in validation techniques, as well as other approaches, for the testing and evaluation of SDS software. Thus the gaps identified here will not necessarily be the most critical, but those which supplement the issues discussed in that report.

1. One of the most critical gaps in validation technology is the need for more experimental work to evaluate the effectiveness of recently developed and proposed research techniques. There has been a flurry of research activity in the last several years leading to a unified and systematic approach to coverage measures, especially with the inclusion of various data flow coverage measures. In a recent study by Weyuker [6], she conducted an empirical study which provided practical and pragmatic data for various data flow coverage measures, where she had previously provided only complexity upper bound measures [5]. More empirical studies of this type are needed, especially using larger programs actually utilized in practice.

2. Nowhere in the research literature does there appear a systematic analysis of regression testing, even though this is a critically important approach for the maintenance of SDS software. A systematic and basic research approach is needed, and yet one which is also pragmatic and can be applied in the short term. The results of such a study should be integrated with other automated support tools.

3. Recently a monograph by Howden [2] has substantially unified the area of functional testing. However, the problem is that there still remain too many aspects of the functional testing approach which depend upon the individual tester's intuition and experience in the selection of test cases. The excellent work of Howden needs to be further extended in order to better automate the process of functional testing.

4. A systematic study of the real cost effectiveness of various validation approaches is needed. An empirical study is needed which not only determines the number of tests required for a specific testing technique or coverage measure, but also models and measures the human (or computer) effort required to select that test data. Further, an analysis and empirical study is needed to determine the <u>mix</u> of testing techniques, and <u>when</u> to apply each during the testing lifecycle for greatest cost effectiveness.

5. For nearly all testing techniques, the lack of a testing oracle substantially increases the cost of the process, and often leads to an ad hoc approach if the tester does not know whether the output of the test data is correct or not. A solution to this problem requires the automation of the decision about correctness. One approach to this problem was provided by the results of Gannon, McMullin and Hamlet [1], in which test data is devised so as to contrast the program and specification, and thus the specification can be viewed as an oracle in this situation. Currently, research by

Lawrynuik and White [3] have the objective to extend this work by also automating test data selection. This is, of course, only one approach to the solution of the oracle problem, other approaches being voting with multiple program versions, etc.

6. It is critically important to design software that is not only correct, but is easy to modify and test during the maintenance phase of the software cycle. Considerable effort has been expended on various solutions to this problem, but another aspect of this issue is to design software so that regression testing can be effectively achieved. A definition of software which is not regression-testable has been made by Leung and White [4], together with a proposal for regression testing for software which is regression testable.

Time Frame and R&D Tasks to Resolve These Deficiencies

For these six deficiencies, I will indicate what could be achieved in the near-term and long-term, and how best to achieve progress (R&D tasks):

1. In the near term we can expect academic researchers to continue research efforts on coverage measures, achieving substantial results, and small-scale experimental work should be encouraged. However, a larger R&D task should be defined in which experimental evaluation of these techniques is funded using larger scale software, and performed by either private industry or an academic research facility capable of conducting large scale software experimental work.

2. In the near term it should be possible to provide a systematic and basic research analysis of regression testing, and a small R&D task should be developed to support this. Since the issues are essentially pragmatic and applied, the results obtained should be quite practical, and amenable to rapid technology transfer. Longer term efforts would then concentrate on experimental studies, matching these analytical results with guidelines and intuition from current practice and experience. Another longer term goal would be the incorporation of these principles into current software design methodology as a major effort of technology transfer.

3. As for the function testing methodology proposed by Howden [2], a near-term goal would be the incorporation and evaluation of this methodology in current software projects which utilize functional testing. It will require a long-term effort to better automate the aspects of functional testing which depend upon the tester's intuition and experience. An R&D task could be defined with this long-term objective in mind.

4. A near-term R&D project can be proposed to model and measure the cost required for various testing techniques or coverage measures; the most difficult issue involves the human effort to select test data. A longer-term project would involve a systematic study which combines a mix of several testing or verification techniques, including the decision of <u>when</u> each might be applied.

5. A continuing problem involving high cost in testing is the unavailability of an automated oracle to determine the correctness of test data. A near-term R&D project involves the development of a prototype system for abstract data types which utilizes the specification as an oracle, and automatically generates test data using PROLOG. It will be a long-term effort to complete this research, and to transfer this to practical software technology for SDS software.

6. A near-term R&D effort should be able to identify the characteristics of software which can be modified and tested during the maintenance phase; this can be done by means of a criterion for regression-testable software. This could then be followed by a longer term project involving the technology transfer of this concept.

Detailed Discussion of My Current Research Work

My current research work can be described as three primary thrusts:

a) One of the serious limitations of domain testing was the necessity of testing all border segments, including the potentially infinite number of border segments due to one or multiple iteration loops. This has been addressed by White amd Wiszniewski [7], in which they have shown that only a finite number of border segments need be tested. Moreover, for the case of linear domained programs, it is indicated how these border segments to test can be selected, and an upper bound is given on the number in that equivalence class which need be tested:

$$1 + (2m/(n-1)),$$

where m is the number of program variables and n is the number of input variables. Although this is quite reasonable, this paper also shows that the number of equivalence classes of border segments to examine can grow exponentially with n, the number of input variables for certain program structures. Since these situations are not unusual in the testing area, the paper also identifies those equivalence classes to first examine in order to derive the greatest information from the testing effort.

Although this represents an important solution to a theoretical problem associated with domain testing, another fundamental and basic problem remains to be solved before domain testing can become generally applicable to SDS software: the approach must be extended beyond the linear case, and shown how it can be applied in a practical way to nonlinear cases encountered in practice.

b) Another research thrust corresponds to a specification-directed software testing system using PROLOG. This is based upon the work of Gannon, et. al [1], in which the specification is represented as algebraic axioms, the program is an abstract data type implementation, and given the specification, the implementation, and the set of test cases, the system provides automated testing. This corresponds to using the specification as an oracle. In the research initially described in reference [3], given the specification and implementation of an abstract data type, the goal is automated generation and analysis of test data using PROLOG. A prototype of this system has been constructed; the essential research issue is to control the PROLOG generation process of test cases so as to choose the test cases providing the greatest information, with a termination condition being provided by appropriate coverage measures.

This research project is related to the near-term R&D task described as 5) in the previous sections. Although the assumptions of algebraic axioms for the specification, abstract data types as the implementation, and the use of PROLOG may not be realistic at this time for SDS software, if this research is successful, the possibility of the specification as an automated oracle would certainly provide an excellent basis for a long-term R&D task to transfer this technology to SDS software.

c) Despite broad reference to regression testing in the testing literature, and its usage in industry, there does not appear to be a systematic study of regression testing as a basic process or technique in the scientific literature. A technical report by Leung and White [4] has provided a systematic description of the problem and classification of various subproblems. Several important concepts have arisen in that report:

- programs which are regression-testable
- corrective regression testing, where the specification does not change
- data structures are defined which support regression testing, most of which can be constructed during the process of test selection to achieve various types of coverage measures
- the two essential problems of regression testing are that of new test data selection and test plan modification

This research aims to solve both the problems of new test data selection and test plan modification, and as such, relates to the near-term R&D tasks described for deficiencies 2) and 6).

References

[1] J. Gannon, P. McMullin and R. Hamlet, "Data Abstraction Implementation, Specification and Testing," ACM Trans. Prog. Lang. Syst. 3.(3). pp. 211-223.

[2] W.E. Howden, Functional Program Testing Analysis, McGraw-Hill Book Co., New York, 1987.

[3] D. Lawrynuik, "The T-3 Testing Tool," Proceedings of the CIPS Edmonton Fall Conference, Edmonton, Alberta, November 16-18, 1987, 6 pages.

[4] H. Leung and L. White, "An Analysis of Regression Testing," Technical Report 88-15, Department of Computing Science, University of Alberta, October, 1988.

[5] E.J. Weyuker, "The Complexity of Data Flow Criteria For Test Data Section," Info. Processing Letters, 19 (2), pp. 103-109.

[6] E.J. Weuker, "An Empirical Study of the Complexity of Data Flow Testing," Proceedings of the Second Workshop on Software Testing, Verification and Analysis," Banff, Alberta, July 19-21, 1988, pp. 188-195.

[7] L.J. White and B. Wiszniewski, "Complexity of Testing Iterated Borders for Structured Programs," Proceedings of the Second Workshop on Software Testing, Verification and Analysis, Banff, Alberta, July 19-21, 1988, pp. 231-237.

[8] C. Youngblut, C. Gonzalez, W. Brykczynski, and J. Salasin, "SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks," Draft IDA Memorandum Report P-2132, August 1988.

Testing Criteria versus Abstraction: Incidental and Inherent Limitations

Steven J. Zeil

Computer Science Dept.
Old Dominion University
Norfolk, VA 23508

Abstract

Despite the important role played by the notion of abstraction in modern methods of software design and implementation, relatively little consideration has been paid to the interaction between abstraction and testing criteria, especially for automatable criteria. A survey of relevant syntactic, semantic, and methodological problems is presented, and a brief overview is presented of research by the author aimed at developing testing criteria free of those problems.

1. Introduction

The concept of abstraction underlies much of software engineering. Abstraction of data and abstraction of function is at the heart of most design and specification methodologies [1,13]. Abstractions are a major feature of any program large enough to be composed of successive layers of modules. Even in cases where we might claim that the abstractions are improper, poorly designed, or incorrect, they still exist and constitute an important characteristic of the code under test. Mechanisms for formally declaring abstractions, such as the subroutine or the Ada package, are provided by most programming languages.

It can be argued, therefore, that a major aspect affecting the utility of any testing criterion is its support for the testing of *abstract code*, which we here define quite conservatively to mean any code containing operations and/or data types that are not provided as primitives in the underlying language. The high-level and medium-level modules in a system cannot be tested by criteria that only provide effective tests when applied to la ,guage-primitive data and operations.

Nonetheless, little attention has been paid to the properties of testing criteria affecting their support for abstraction. This paper will outline a number of properties of current testing criteria that limit their effectiveness with abstract code, and will provide a brief overview of research by the author aimed at developing testing criteria free from these defects.

2. Common Problems in Testing Abstract Code

A survey of testing criteria, especially automatable testing criteria, appearing in the current literature reveals a number of properties that limit the effectiveness of those criteria when applied to abstract code. In many cases, these limitations are incidental — they can be relieved by relatively minor changes prompted by a heightened awareness of the need to deal with abstract code. In other cases, the limitations are inherent, stemming from the basic nature of the criterion and therefore not easily alleviated. These properties can be divided into three classes: syntactic, semantic, and methodological.

2.1 Syntactic Problems

Syntactic limitations arise when a testing criterion treats code operations differently according to the syntax in which they are written. Given that most languages permit many syntactically different ways of writing the same code, this can lead to wildly different tests for semantically identical code. Furthermore, since the syntactic options may vary with the level of abstraction, such syntactic dependencies by a criterion can interfere with its ability to work at different levels of abstraction.

As an example of the problems that can arise from syntactic dependencies, consider three different ways of writing the union of two sets to form a third:

```
Result := Set1 + Set2;

Result := Union(Set1, Set2);

Union (Set1, Set2, Result);
```

The semantics is the same in each case. There seems to be little reason that the test data resulting from an examination of these alternative codes should differ[1]. Most testing criteria, however, should show significant differences in their treatment of these cases. Most mutation testing systems [3,5], for example, would treat all three differently, especially if "+" were language-defined and "Union" were user-defined. Hamlet's expression coverage criterion [9] would treat the operator and function cases identically, recognizing them both as simple expressions, but would treat the procedural case quite differently. Among other testing criteria, and ignoring trivial cases such as path selection criteria that completely ignore statement semantics, perhaps only Howden's functional testing [11] strategy would treat all three cases identically, recognizing each form of the union as an instance of a "design function".

A common source of syntactic dependencies is the fact that certain of the operations made available by a programming language are left implicit or are denoted by a distinct syntax, making it tempting to ignore their existence or to treat them as special cases when formulating a testing criterion. Examples of such operations include type conversions, often not shown explicitly in the source code, and the array subscripting and record component selection operators, usually written in a syntax different from that of other operators. Consider the example of a program to solve a linear system of equations. The designer of this program may choose to store the coefficients in a two-dimensional array, accessed by operations like "Coeff[I,J]", or in a more elaborate structure, such as a sparse matrix structure accessed by user-defined operations such as "GetCoeff(I,J)". Clearly, the lower-level codes to create and fill these structures may differ, and consequently the test cases for those codes should differ. At the levels where the code is identical except for the choice of "Coeff[I,J]" versus " GetCoeff(I,J)", however, the test requirements should be the same, if only to allow for a possible change of underlying representation later in the lifetime of the program. It is rare, however, for any testing criterion to recognize the semantic similarity between these two alternative codes.

1. It can be argued that syntactic considerations have a place in determining the choice of test data, in that the syntactic quirks of a particular language may contribute to the chances of introducing certain faults. A problem with this argument, however, is that the conditions under which a test may reveal the presence of a given fault depend as much, and often more, upon the context in which the fault occurs as upon the nature of the fault itself [14,20]. To the degree that the argument is valid, however, semantic considerations would still argue that a substantial portion of the test data should be free of syntactic dependence.

Syntactic limitations are, as a rule, not inherent in a testing criterion but can be easily relieved by focusing attention upon the abstract syntax of the code under test rather than upon the concrete syntax.

2.2 Semantic Problems

Semantic problems arise when a criterion is designed to deal with a fixed set of data types and operations. Almost every testing criterion is based upon some view or model of the semantics of the code under test. Inherent semantic problems arise when these semantic models clash with the actual data types and operations appearing in the code. Incidental semantic problems usually stem from unnecessary limitations on the model. Two forms of semantic problem will be discussed here: semantic bias and locality.

We will say that a criterion is *semantically biased* toward lower levels of abstraction if the effectiveness of the strategy decreases sharply as the level of abstraction of the data and the operations in the code under test increases.

In the most severe cases of semantic bias, the testing strategy cannot even be applied to software containing unanticipated data types and operations. For example, algebraic testing [10] and transcendental selection [15] view the program as a black box computing a numerical function of numerical inputs. Tests selection is based upon algebraic properties of the class of functions in which the program is supposed to lie (e.g., choosing any *n* distinct inputs for a program expected to compute a polynomial of degree *n* on a single input). These criteria would be essentially useless with a program whose inputs and outputs were, for example, character strings.

Some testing criteria with similar semantic models exhibit a less severe form of semantic bias, in which a criterion can be applied to software having unanticipated data types or operations, but the criterion essentially ignores the portion of the code incompatable with the model. Examples of this level of bias are afforded by domain testing [4,16] and perturbation testing [20], which test numerical portions of program code, but generate no test data for portions of the code involving non-numeric operations or even numeric operations without the desired algebraic properties.

These inherent limitations are usually quite deliberate and should not be considered deficiencies. By taking advantage of a body of knowledge about specific algebraic systems, these criteria are able to offer more rigorous testing than can be expected in more general domains. For the purposes of this discussion, however, it is important to realize that, because much software lies outside of those restricted algebraic domains, we must have reasonable testing criteria that are not limited to those domains. The examples of bias discussed above are not defects, but they are limitations.

In many instances, however, semantic bias represents incidental limitations that could be alleviated to obtain more generally useful criteria. For example, in [7], a rule is given requiring, for any expression of the form "X **relational-op** Y", where X is a variable and Y is a variable or a constant, that X be given a value that "differs from the constant [Y] by the smallest decrement (increment) allowed." The same author later [6] reworded this rule as "variable1 - variable2 (or constant) = 0, $+\epsilon$, and $-\epsilon$, *where* ϵ is the smallest expressible value in the variables' domain." This latter version of the rule is far less general because it is restricted to data for which subtraction, unary negation, and the constant 0 are defined. The original rule could be applied to any data type of finite domain for which a strict ordering existed (e.g., characters or enumerated types). The assumption of unnecessary operations is a common source of semantic bias.

Similar incidental limitations arise in mutation analysis, where the practice of defining the mutations in terms of the language primitives results in many mutations useful only at the primitive levels of abstraction. Common mutations such as "if the variable X appear in some statement, replace it by "X+1" are limited because "+" is only defined for certain data types, and "+1" is defined for even a smaller set of types. The danger here is that this rule hints at but does not capture a more general testing principle that is as valid for many abstract types as it is for integers. For example, just as we would want to test the conditional statement in the code

```
        :
        I := I + 1;
  end loop;
  if I >= N then
        :
```

for "off-by-one" errors, so we would also like to test the conditional statement in the code

```
        :
        Q := insert_in_queue (Q,X);
  end loop;
  if queue_is_full(Q) then
        :
```

for "one-too-many-insert" errors. This requires, however, an unusual degree of freedom from semantic bias.

Another form of semantic problem is *locality*, the degree to which the testing requirements imposed by a criterion at a particular code location are based only upon the code near that location[2]. Locality is a limitation, in the sense that it serves to restrict the amount of information that must be considered by a testing criterion. For example, many testing criteria seek to force program variables to take on distinct values from one another. Criteria such as mutation testing, weak mutation testing [12], and expression coverage exhibit locality when they require X to take on a value distinct from Y only at those locations where one of those two variables already appears in the code.

Although a certain degree of locality may be useful in making an otherwise intractable goal possible, absolute locality is undesirable, in part because errors of omission tend to remove the context that would enable a local criterion to formulate revealing tests. This property is of particular relevance to abstract code, because one of the effects of abstraction is often to hide the underlying data manipulation. Consider, for example, the problems posed for local criteria in testing the code:

```
while not Transaction_EOF() loop
   Read_Transaction_Record;
   Match_Master_File;
   Apply_Transaction_To_Master;
   Write_Master_Record;
end loop;
```

2. If we regard the execution state of a program to be characterized by (L,V) where L is a code location and V is a sequence of values, one for each variable in the program, then almost every implementation-based testing criterion can be considered to impose testing requirements of the form "for some test, reach a state (L_1,V) such that $P(V)$ is true". Thus we can speak of testing requirements imposed "at a location".

The only plausible way of testing this code is to use non-local information (e.g., requiring that the keys of the master and transaction records be tested when already matching and when not matching, that each file be tested both at and before the end-of-file, etc.).

2.3 Methodological Problems

The various abstractions present in a module represent a statement from the designer as to what information is required to understand that module and, especially when enforced by a language-provided encapsulation facility, what information is legal as an aid to that understanding. A testing criterion that violates the designer's intent risks the formulation of test requirements that will not be easily understood by the testers. Even worse, the choice of test data may depend upon details of the underlying representation, details that are subject to later change, resulting in the unpleasant scenario of having to choose substantially new test data for modules that directly or indirectly call a recently changed module, even though the changes were "invisible" at the higher level of abstraction.

It is apparent, therefore, that a testing criterion should be "discreet", meaning that it "only looks where it is supposed to look". A testing criterion that requires the tester to understand details from lower levels of abstraction than are manifested by the code under test or that requires the application of operations that are not a visible part of the abstractions available to that code will be termed *indiscreet*. A certain amount of indiscretion may be tolerable. For example, the need to test the consistency of interfaces between successive levels of abstraction may require knowledge from both levels, but such indiscretion should be limited to certain designated tests and should not span multiple levels of abstraction.

Indiscretion in testing criteria is manifested in a number of ways. It can appear as an emphasis upon the structure underlying an implementation, as in [11], where several rules for arrays are stated that are actually more properly associated with the mathematical concept of matrices. These rules would be appropriate even if the underlying implementation of the matrices were changed (e.g., to a sparse representation). In mutation testing and expression coverage, indiscretion arises from the practice of limiting the application of operators only by the legality of the resulting expression in the programming language. In situations where the language does not provide (or the programmer does not employ) sufficient control over encapsulation and the inheritance of operations (or where the programmer has not made use of the available language features), this can lead to such indiscretions as incrementing a "pointer" to a node of a linked list that just happens to have been implemented as an array indexed by integers.

Side effects of called modules represent a particular problem in maintaining discretion. For example, many criteria require the ability to monitor the output of a function or procedure call. Suppose, then, that a module M_1 with a single output parameter of type T_1 has, as a side effect, the property of assigning a value of type T_0 to some global variable. Suppose further that another module M_2 calls M_1. In testing M_2, we might be tempted to claim that M_1's "real" output is $T_1 \times T_0$ and to adjust our testing criterion accordingly. The problem is that the type T_0 might not be visible in M_2, making such an adjustment an indiscretion. Without the adjustment, however, many testing criteria will be significantly weakened. As we move up in successive layers of abstraction, from M_2 through M_3 to M_k, the indiscretion associated with knowing that a call to M_{k-1} results in a manipulation of a value of type T_0 becomes progressively more serious. It is not surprising, therefore, that few published testing criteria have specified the treatment of such side effects.

3. Testing Criteria for Abstract Code

Relatively few testing criteria have been formulated with abstract code as a major concern. Exceptions of note are the DAISTS system [8], which required every expression in the code to have been evaluated to yield at least different results, and the specification-based testing criterion by Bouge et al. [2], which uses logic programming languages to iterate over the structure of recursively-defined data types, thereby producing a sequence of abstract data values for use as test inputs. Also of note is Howden's functional testing [11], which provides an organizational framework within which testing of abstract code could proceed, although the choice of appropriate criteria to apply within that framework for abstract code is still unclear.

My own work in this area has concentrated upon implementation-based techniques that seek to avoid the problems outlined above. There are currently two thrusts to this work, one a coverage method called EQUATE and the other a fault-based method called symbolic fault tracking.

3.1 EQUATE

EQUATE [19,17] is a coverage method based upon two very simple ideas:

1. each object manipulated by the code should take on at least two distinct values during testing;
2. each pair of objects manipulated by the code should take on values distinct from one another during testing.

These ideas are hardly new, having been incorporated into a wide variety of previous testing criteria. The distinctive features of EQUATE may be found in the elaboration of these ideas:

- These ideas are applied as a uniform coverage metric at every statement (not just at the statements where the relevant objects are manipulated) to minimize locality;
- The phrase "objects manipulated by the code" is interpreted in a fashion that avoids syntactic dependencies; and
- The meaning of "distinct" is chosen in a manner that avoids semantic bias, maintains a high level of discretion even for languages/programs in which not all abstractions are formally declared and encapsulated, and that yields challenging test requirements for a wide variety of data types and operations.

Of particular note is the fact that "distinct" is a *stronger* property in most cases than would be the usual formal definition of "not equal", as one can easily show common examples of abstract objects for which the use of conventional inequality as the test for distinction results in trivial testing requirements [19].

A simple prototype EQUATE system has been implemented, and implementation of a more complete version (based upon the ARIES interpretation system [21]) should begin soon. Once constructed, that system will be employed in an empirical study to compare various design options remaining within the EQUATE strategy to see which options yield the most effective tests. Subsequently, a study comparing the effectiveness of EQUATE with other testing methods from the general literature is anticipated.

3.2 Symbolic Fault Tracking

Perturbation testing [20] was developed for the testing of numeric code. It's key idea was the introduction of a "variable" denoting a potential fault in an expression and the subsequent derivation of conditions under which that variable could be non-zero and yet not affect the program state resulting from execution of that statement, i.e., conditions under which a fault could be present but not originate an error. A similar approach by Morell and Hamlet later focused upon the derivation of conditions under which such a fault could be present without the change of state so engendered being propagated to the program output [14]. A somewhat different approach to dealing with propagation was later added to perturbation testing [18]. None of these advances moved outside the realm of numeric code, but the basic idea of introducing a symbolic fault and using symbolic execution to generate the conditions under which that fault would go undetected by the current test set would seem to be applicable to more abstract code.

I have recently begun investigating the use of a combination of techniques from my own perturbation work and from the work of Morell and Hamlet in more abstract domains. There appears to be promise of a tractable technique based upon generating a system of constraints describing the circumstances under which a symbolic fault would not manifest as an actual error. Such a system of constraints would contain the fault as one parameter, with the other parameters being a set of inputs to the code under test. The demonstration that this set of constraints were unsatisfiable would imply that no fault (within some user-specified complexity level) could exist at that location without detection. Satisfiability would, on the other hand, indicate the need for further testing and the choice of a test set from outside the solution set of the constraints would serve to further reduce the number of faults that could exist without detection by the accumulated test sets.

If successful, the ability to track symbolic faults could prove useful as a means of evaluating testing criteria, as a criterion in its own right, or as analytic aid to other criteria (e.g., to allow "pruning" of a mutation set by proving that a given location need not be mutated because any mutation-like faults would, if present at that location, have already been caught).

REFERENCES

[1] G. Bergland. A guided tour of program design methodologies. *IEEE Software*, 14(10):13-37, Oct. 1981.

[2] L. Bouge, N. Choquet, L. Fribourg, and M. C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343-360, Nov. 1986.

[3] T. A. Budd. *The Portable Mutation Testing Suite*. Technical Report TR 83-8, University of Arizona, March 1983.

[4] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at Domain Testing. *IEEE Transactions on Software Engineering*, SE-8(4):380-390, July 1982.

[5] R. A. DeMillo, F. G. Sayward, and R. J. Lipton. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34-41, Apr. 1978.

[6] K. A. Foster. Comment on the application of error-sensitive testing strategies to debugging. *ACM Software Engineering Notes*, 8(5):40-42, Oct. 1983.

[7] K. A. Foster. Error sensitive test case analysis (ESTCA). *IEEE Transactions on Software Engineering*, SE-6(3):258-264, May 1980.

[8] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211-223, July 1981.

[9] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279-290, July 1977.

[10] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10:53-66, Oct. 1978.

[11] W. E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, SE-6(2):162-169, March 1980.

[12] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(2):371-379, July 1982.

[13] B. Liskov and S. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, SE-1(2):9-19, March 1975.

[14] L. J. Morell and R. G. Hamlet. *Error Propagation and Elimination in Computer Programs*. Technical Report 81-1065, University of Maryland, July 1981.

[15] J. H. Rowland and P. J. Davis. On the use of transcendentals for program testing. *Journal of the ACM*, 28(1):181-190, Jan. 1981.

[16] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, SE-6(3):247-257, May 1980.

[17] S. J. Zeil. Complexity of the equate testing strategy. *Journal of Systems and Software*, 8(2):91-104, March 1988.

[18] S. J. Zeil. Perturbation testing for domain errors. To appear in *IEEE Transactions on Software Engineering*.

[19] S. J. Zeil. Testing for equivalent algebraic terms – EQUATE. In *Proceedings of the ACM SIGSOFT/IEEE Workshop on Software Testing*, pages 142-151, IEEE, July 1986.

[20] S. J. Zeil. Testing for perturbations of program statements. *IEEE Transactions on Software Engineering*, SE-9(3):335-346, May 1983.

[21] S. J. Zeil and E. C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the Tenth International Conference on Software Engineering*, pages 241-248. IEEE, Apr. 1988.

Panel: Formal Verification

R. Meeson, Chair

Panelists:

R.A. Kemmerer
T.E. Lindquist
M.K. Smith

Critical Gaps in Formal Verification Technology

A Position Paper

Richard A. Kemmerer

Department of Computer Science
University of California
Santa Barbara, CA 93106

The Strategic Defense System (SDS) is a large distributed system, which requires timely coordination between its components. This coordination puts severe real-time constraints on most of the software components. Given the critical nature of SDS and the catastrophic consequences that will result if it fails to perform appropriately, it is imperative that its reliability be guaranteed before it is deployed.

Formal specification and verification techniques have become an accepted approach to achieving reliable software. Today there are a number of formal verification systems available [AK85,CDL85,GDS84,SAM86,TE81] all of which use mathematical techniques to guarantee the correctness of the system being designed and implemented. To use these techniques it is necessary to have a formal notation (usually an extension to first-order predicate calculus) and a proof theory. Unfortunately, none of these systems are capable of specifying and verifying the critical real-time requirements of a distributed system like SDS.

In this paper some critical gaps that exist in the formal verification technology base with respect to the Strategic Defense System are identified. Long and short term solutions to close these gaps are also presented.

Critical Gaps

A major gap in the formal verification technology is the *lack of production quality tools*. In a recent study [Kem86] four verification systems were thoroughly examined by experts in the field. The verification systems reviewed represent the leading edge of mechanical verification technology. This mechanical support is useful, if not necessary, when applying formal verification to real applications. However, each of these systems has been built primarily as a research vehicle for exploring different ways of implementing and applying formal verification. None of them has been designed or implemented as the kind of production quality system that is needed to support wide-spread application of verification to real software systems and in particular to a large distributed real-time system like SDS. There is much that needs to be done to progress from where the systems are now to a truly production quality verification system.

A production quality verification system must be well engineered. It needs to have a high quality user interface. It must perform efficiently. It must be robust, well documented, maintainable, etc. It should be built with the best methods available for software engineering, quality control, configuration management, etc. Generally, these issues have not been given a high priority in the implementation of the research prototypes, and all of them have major deficiencies in some of these areas. The current systems have been developed primarily to demonstrate the feasibility of i) mechanizing formal verification and ii) applying it to real software systems. They have served that purpose well, but they are far from being production quality systems.

If a production quality system is to be made available on a wide-spread basis, *education of the potential user community* will also be required. Most software developers are frightened by the prospect of having to use formal verification tools. It is not that they do not possess the native intelligence to comprehend formal specifications, but rather that they have not received the appropriate training. Thus, the gap is an education gap. The software development community needs to be educated in the fundamentals of verification as well as in the use of mechanical verification tools.

The complexity of writing correct programs increases with the introduction of parallelism and is further complicated with the introduction of real-time constraints. Similarly, the difficulty of formal specification and verification increases from sequential to parallel to real-time programs. Like sequential programs, real-time programs are judged against critical correctness conjectures. Real-time systems, however, must also meet critical performance deadlines. Therefore, the verification of real-time systems involves demonstrating that the specified system meets the performance deadlines in every case and, in particular, in the worst case.

The currently available formal verification languages and systems can be used to specify functional properties of sequential systems (and some limited parallelism), and to verify that the specifications satisfy the desired critical requirements. None of these systems, however, can be used to *specify and verify performance requirements*. This is another major gap in the formal verification technology that will have to be bridged before there can be any hope of verifying SDS software.

Near Term Solutions

The following are suggestions for near term (2 to 3 years) solutions to the critical gaps in formal verification technology that were identified in the previous section.

Production Quality Systems

Two to three years is not enough time to achieve a production quality formal verification system. However, it would be useful to enhance the existing formal verification systems to bring them closer to production quality.

Education

The existing research prototype systems can play a useful role in educating the software development community, and they can be used to explore a wider variety of applications of formal verification. Demonstrating the effectiveness of verification on an increasing variety of important applications probably is the best way of drawing the attention of the software engineering community to verification, and thereby accelerating its development.

Verification of Performance Requirements

A design for a formal specification language and proof methodology for specifying real-time systems currently exists. The language is RT-ASLAN, which is an extension of the ASLAN specification language. RT-ASLAN supports the specification of parallel real-time processes through arbitrary levels of abstraction. RT-ASLAN allows concurrent processes to be specified as a collection of subspecifications. The kind of systems that can be specified are loosely coupled systems communicating through formal interfaces. It is also assumed that systems specified in RT-ASLAN have a processor associated with each subspecification. Therefore, the systems that can currently be specified using

RT-ASLAN are neither process restricted nor resource restricted.

From RT-ASLAN specifications performance correctness conjectures are generated. These conjectures are logic statements whose proof guarantees the specification meets the desired critical time constraints. More details on the language and the correctness conjectures, as well as example RT-ASLAN specifications, can be found in [AK86].

The design for the extensions to the ASLAN language processor that will allow it to accommodate RT-ASLAN are complete. A reasonable near term solution would be to implement the extensions to the ASLAN language processor. When the current version of the RT-ASLAN language has been implemented RT-ASLAN can be applied to real-world real-time systems to evaluate its usefulness.

Long Term Solutions

The following are longer term (ten or more years) solutions to bridge the technical gaps in the formal verification technology that were identified in the first section.

Production Quality Systems

To achieve a truly production quality formal verification system it will be necessary to build a new system rather than attempting to enhance the existing systems. The verification assessment study that was performed for the National Computer Security Center [Kem86] identified some components that should be included in a state-of-the-art formal verification system. They are:

- a specification language based on first-order, typed predicate calculus,
- an imperative language derived from the Pascal/Algol 60 family of programming languages,
- a formal semantic characterization of the specification and implementation languages,
- a verification condition generator (VCG),
- a mechanical proof checker with some automated proof generation capabilities,
- a (small) supporting library of (reusable) theorems,
- a glass (as opposed to hard copy) user interface, possibly using bit mapped displays,
- a single system dedicated to one user at a time (e.g., a workstation dedicated to the verification process), and
- the embedding of these components in a modest programming environment with utilities such as version control.

In the verification assessment report some near term advances that should be included in a next generation formal verification system were also identified. They were:

- improvements in system interfaces through the use of graphics interfaces (as exemplified by bit mapped graphics workstations),
- improvement in theorem provers so that larger inference steps are possible,
- improvement in the expressibility of specification languages,
- the development of reusable theories, and
- the application of formal semantics to increase confidence in the programming and specification languages and in the reasoning systems.

Education

The long term solutions to bridge the education gap are to continue with the near term solutions. As new improved formal verification technologies are developed software developers should be educated in the use of the new technologies and tools.

Verification of Performance Requirements

RT-ASLAN provides a means of formally specifying and verifying a restricted class of real-time systems. However, further research could extend the RT-ASLAN language and proof methodology to allow the formal specification and verification of a less restricted class of real-time systems. There are a number of areas that need further investigation. In particular, low (almost implementation) levels of specification must be investigated more closely. It is easy to state that "communicating transitions call interface transitions", or that "TIME is an implicit variable representing a clock"; however, these concepts are likely to be hard to implement in a way that fits into the RT-ASLAN formalisms.

Another area that requires further research deals with processor and resource availability assumptions. The preliminary design for RT-ASLAN can be used to specify only systems that have plenty of processors and plenty of resources. RT-ASLAN will need to be modified or extended to deal with systems that are either process restricted or resource restricted.

Real-world systems frequently consist of several virtual-processes (i.e., cyclic transitions) sharing a single processor. The processor sharing arrangement is commonly based on the priorities of the virtual-processes. Harter [Har84] describes a "time dilation" technique for arriving at upper and lower performance bounds for priority level systems under the following two constraints: virtual processes are cyclic and run either until completion or until preempted by a higher priority virtual process, and virtual processes are allowed to make procedure calls only to higher priority virtual processes. Harter's time dilation algorithm and the syntax necessary to support it are likely candidates to be added to RT-ASLAN to allow the specification of priority level systems. This would allow more realistic systems to be easily specified.

Conclusions

The three gaps in the formal verification technology base that were discussed in the previous section are just some of the gaps that exist. Time did not allow for expansion on the other deficiencies. Two that come immediately to mind are executable specifications and formal verification of encryption protocols.

Given the large size, distributed nature, and extreme real-time constraints of the proposed Strategic Defense System it is unrealistic to assume that testing techniques will suffice as assurance measures to guarantee the reliability of the SDS software systems. Therefore, it is imperative that formal verification techniques be used. However, given the state of the current formal verification technology base it is also not a candidate for assuring the high degree of reliability necessary for SDS software. Clearly, more research in this area is necessary.

References

[AK85] B. Auernheimer and R.A. Kemmerer, "ASLAN user's manual", Department of Computer Science, University of California, Santa Barbara, TRCS84-10, March 1985.

[AK86] B. Auernheimer and R.A. Kemmerer, "RT-ASLAN: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9, September 1986.

[CDL85] J. Crow, D. Denning, P.Ladkin, M. Melliar-Smith, J. Rushby, R. Schwartz, R. Shostak, and F. von Henke, "SRI verification system version 2.0 user's guide," SRI International Computer Science Laboratory, Menlo Park, California, November 1985.

[GDS84] Good, D.I., B.L. DiVito, and M.K. Smith, "Using the Gypsy methodology," Institute for Computing Science, University of Texas, Austin, Texas, June 1984.

[Har84] Harter, P.K. "Response times in level structured systems", Department of Computer Science, University of Colorado at Boulder, CU-CS-269-84, July 1984.

[Kem86] Kemmerer, R.A., *Verification Assessment Study Final Report, Volumes I - V*, C3-CR01-86, National Computer Security Center, Fort George Meade, Maryland, January 1986.

[SAM86] Scheid, J., S. Anderson, R. Martin, and S. Holtsberg, "The Ina Jo Specification Language Reference Manual," SDC document, System Development Corporation, Santa Monica, California, January 1986.

[TE81] Thompson, D.H. and R.W. Erickson, eds, "AFFIRM reference manual," USC Information Sciences Institute, Marina del Rey, California, February 1981.

Computational Logic Research Status

Michael K. Smith

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703

1. Introduction

Research at Computational Logic Inc. (CLI) is currently focused on two major projects, Rose and Trusted Systems, that address the formal verification of software and/or hardware. Both are based on the Boyer-Moore Logic and Theorem Prover. Other work in progress at CLI consists of maintenance and development work on the Gypsy Verification Environment. In the next three sections we summarize the status of verification technology at CLI and then discuss ways in which it might be applied to address some of the problems facing Strategic Defense System software and ways in which this application might be accelerated. The primary points made in these last sections are:

- DoD has a critical need to build mathematics into the thinking of the community that is developing software for SDS. As systems get more complex and life-critical the need for objective measures of extreme reliability increases.
    1. We need an emphasis on *ends*, not *means*. In particular, the policy setting organizations need to promulgate a shared vision of adherence to precise and sound reasoning that will filter out half-baked and unsound approaches and encourage the adoption of mathematical thinking.
    2. We need experience with worked examples.
    3. We need to enlarge the pool of technical talent capable of applying formal methods to software development.
    4. Note that none of this requires that automatic tool support exist. While this is desirable it is secondary to a community mindset that sees software development as a highly mathematical activity.
- For SDS, code proof may be of more importance than design proofs. Consider two possibilities:

Scenario 1:	A top level specification of SDS has been written and proved to satisfy some properties.
Scenario 2:	The real-time behavior of 10,000 lines of assembly code that are of critical importance to the proper operation of SDS has been specified and the code has been proved to satisfy the specification.

Scenario 1 provides some useful guidance. However the assurance that it provides is extremely divorced from any possible implementation. Scenario 2 provides a much more rigorous statement about a limited, but very specific component.

- There is no hard and fast line between hardware and software. Proofs about hardware are possible. Specifications about hardware (what does this instruction set *really* do in all cases?) are critical to making extremely strong predictions about a system.

2. Rose

[Boyer&Moore79] describes a quantifier free first-order logic and a large and complicated computer program that proves theorems in that logic. The major application of the logic and theorem prover is the formal verification of properties of computer programs, algorithms, system designs, etc. Theorems proved using the theorem prover include the invertibility of the RSA Public Key Encryption Algorithm [Boyer&Moore84], the Turing completeness of Pure Lisp [Boyer&Moore83], and Goedels' incompleteness theorem [Shankar87]. (See also Section 3.)

A complete and precise definition of the logic can be found in Chapter III of [Boyer&Moore79] together with the minor revisions detailed in section 3.1 of [Boyer&Moore81].

Rose is being developed on top of the Boyer-Moore Logic. Initially it has simply *been* the Boyer-Moore Logic. Rose is a language for describing mathematical functions and proofs of properties about them. The language has a precisely defined set of axioms and rules of inference which are implemented in a mechanical proof checker. This proof checker acts as a fully trustworthy mechanical colleague with whom one can reason about Rose functions – a colleague that accepts only valid reasoning. Thus, the Rose language and its proof checker provide a powerful way of developing mathematical theories.

Many functions in Rose can be compiled into efficient programs for digital computing systems. Thus, the full mathematical power of Rose also can be used to specify, implement and prove properties about computing systems. One can reason, both manually and mechanically, about computing systems in the same way as any other function. This provides a rigorous, mathematical approach to predicting the behavior of computing systems.

The major elements of a Rose Development Environment, as we currently see it, are as follows:

- **Functional Language.** Throughout all stages of its life cycle, both the software system and its specifications will be expressed in a precise, well-defined functional language that is based on a formal logic. This logic can be used as a basis for deductive reasoning about the software system.
- **Journal Level Proof Checking.** A key component of the deductive software development system will be an automatic proof checker which can be used to reason about the software under development. Many parts of the system, such as the optimizing compiler and the database manager, will ask questions of the proof checker while manipulating code and specifications. But the primary use of the deductive component will be to provide the user with a mechanical colleague capable of following arguments about the evolving software and debugging his claims and intuitions. We believe that most of these arguments can be carried on at about the level that computer scientists normally use in technical journals.
- **Reusable Theories.** The effectiveness of this deductive reasoning approach will depend in large measure on how much reasoning will need to be performed for a particular software system. This will be minimized by the development of libraries of reusable theorems. Although the development of these theories may be very difficult, we have evidence that they can be developed; and once they are developed, they will

- greatly reduce the cost of the deductive reasoning because it will be possible to reason from very powerful previously proved theorems rather than from primitive axioms.

This deductive software development environment will provide an innovative software technology. The general idea of using deductive reasoning in support of software development is not a new one. Basic ideas about incorporating deductive reasoning into the software life cycle appear in [Dijkstra68], and a much more recent review appears in [Hoare82]). However, these ideas have yet to be molded into a viable technology for wide-spread use.

Previous research at Computational Logic and The Institute for Computing Science at The University of Texas at Austin has passed major milestones in mechanized deductive reasoning that we believe indicate that it will be possible to apply these basic ideas on a wide-spread basis. First, several convincing examples of mechanical deductions about important, real software systems have been successfully completed [Good82a, Good82b] – despite the fact that the software was written in a programming language less supportive of deductive reasoning than that described here and despite the fact that the mechanical proof checker required much more user guidance. Second, the capability of mechanically following journal level proofs has been demonstrated [Boyer&Moore83, Boyer&Moore84, Russinoff83]. These milestones are the basis of our belief that the deductive software development system we are constructing can make deductive reasoning about real software systems economically viable. In particular, we believe that this new technology, as compared to current technology, will eventually enable the attainment of much higher quality software systems for a smaller, and possibly much smaller, investment in time and money.

2.1 Computing as a Physical Science

Donald Good has been developing a view in which computing takes its place as a branch of the physical sciences. Seen in this light, the use of formal specifications of computer systems is comparable to the use of familiar physical laws. These points are amplified in [Good88a], presented at Compass '88, and in "Computing is a Physical Science," which is to be presented as the keynote paper for VDM 88 in Dublin in September, 1988. These ideas will also be presented in an address to the US National Academy of Sciences in September.

2.2 Reusable Libraries

We have developed what we think is a reusable library of definitions and lemmas related to hardware specification and proof. The library consists of about 100 definitions in the theories of natural numbers, integers, lists, and bit vectors. Some 900 lemmas are proved about these definitions, including the correctness of some metatheoretic simplifiers for arithmetic expressions. A CLI note describing the library is also available [Bevier88].

The library is being used as part of the Core Mips specification effort (see 3.1-B) and will see further use when we start to prove properties of that specification. It will take many such applications to convince us that the library is really reusable. (It is common in all rule-based systems for the developers of the rule data base to be fooled into thinking their rules are more general and powerful than they really are. On trials other than the one for which the data base was developed the rules are often seen to be limited and directed only toward the kinds of problems that arose in their development.)

2.3 Rose Common Lisp

We have settled on a preliminary design for the next version of the Rose programming language. There were four design criteria. First, we wanted the language to be efficient enough to be practical. Second, we wanted it to be powerful enough so that we could use it in our own work (thereby giving us more experience with our own tools). Third, we wanted the language to be naturally formalized within the existing Boyer-Moore logic so that proofs about programs in the language were also natural. Fourth, we wanted the language to be easy to "sell" to an outside community. We believe we can formalize a subset of Common Lisp that meets these four criteria.

We are currently working on a prototype "translator" that translates a system of Common Lisp variable declarations and procedure definitions into a set of functions in the Rose logic. The idea is that applicative programs in Common Lisp should translate with almost no change in their appearance into functions in the logic. Non-applicative programs will undergo a more noticeable transformation, but will not necessarily become "unnatural." For example, a Common Lisp program that fails to be applicative because it is sensitive to some special or global variable may be translated to a function with that variable as an additional parameter. A program whose only non-applicative effect is to change the value of some special or global variable may be translated into a pair of functions, one of which returns the value of the program and the other of which returns the final value of the effected variable. A statement about the value or effect of a Common Lisp program will undergo an analogous translation, relative to the conventions established by processing the Common Lisp system, into a formula of the logic so that if the formula is a theorem, the Common Lisp program behaves as stated. We believe that if the translation to the logic is carried out with due consideration of a dozen or so special cases, the resulting logic definitions will be manageable, both intellectually and mechanically.

We are currently focused on modeling error handling, special and global variables and the use of property lists. These features of Common Lisp account for the vast majority of non-applicative programs in the Boyer-Moore theorem prover. Our goal is to handle a sufficiently large subset of Common Lisp that the theorem prover can be coded acceptably in it. This will, of necessity, require the eventual formalization of file I/O, nonlocal exits, and destructively modified arrays, as well as the issues of current concern. We believe that this subset is sufficiently large and rich (and yet sufficiently close to the existing logic) that the Rose verification system will be a very attractive tool to the Common Lisp programmer.

2.4 Concurrency

Our recent work on the mechanical verification of concurrent programs has investigated the specification of non-deterministic transitions and the definition of predicates that define the interesting properties of concurrent programs. The mechanism for specifying non-deterministic transitions and the definitions of these predicates has been done in the Rose logic. The general goal of this work is to formalize the Unity system of Chandy and Misra [Misra81] in the logic.

A concurrent program is a set of relations from states to states. An execution of a concurrent program is a sequence of states, where each consecutive pair of states (a transition) is possible under some relation in the program. Each relation in the program is specified as a boolean valued function of two arguments: the previous and next states. Allowing non-deterministic transitions simplifies the statement of many concurrent algorithms.

The definition of predicates defining the interesting properties of concurrent programs simplifies both the statement and proof of program correctness. The predicates capture basic notions of partial correctness (invariance) and total correctness (liveness). They also function as proof rules and therefore guide the verifier when proving program correctness. These predicates are based on others described in the literature of distributed systems.

Invariance is captured by a predicate, UNLESS. UNLESS(P, Q, Prg) specifies that in the program Prg, if P is true about any state in the program execution, P continues to be true about every subsequent state until Q is true. The most limited notion of liveness is specified by a predicate, ENSURES. ENSURES(P, Q, Prg) specifies that in the program Prg, there exists a single transition such that for all states satisfying P, the next state satisfies Q. A more general notion of liveness is specified by taking the transitive closure of ENSURES.

3. Trusted Systems

Previous applications of the Boyer-Moore Logic and Theorem Prover [Boyer&Moore79] include:

- A mechanical proof of the Turing completeness of Pure Lisp [Boyer&Moore83].
- A proof of the invertability of the RSA Public Key Encryption Algorithm [Boyer&Moore84].
- A proof of Goedel's Theorem [Shankar87].
- A verified operating system kernel [Bevier87a, Bevier87b].
- A verified assembler [Moore88].
- A verified microprocessor [Hunt85, Hunt87].

The existence of Warren Hunt's verified design for a general-purpose processor clearly suggested the idea of using that processor as the *delivery vehicle* for some *trusted system* such as an encryption box, a verifiable high-level language, or perhaps even a program verification system. This was an inevitable suggestion since Hunt's FM8501 was developed in the same laboratory responsible for the Gypsy Verification Environment [Good88b] and the Boyer-Moore theorem prover [Boyer&Moore79, Boyer&Moore88], two of the most widely used program verification systems in the world. But, unless one wants to build such applications in machine language, it is necessary to implement higher-level languages on FM8501. To maintain the credibility of the system, the implementation of those languages should be verified all the way down to the FM8501 machine code.

The Trusted Systems effort is an attempt to apply mechanical verification technology to the construction of such hierarchically constructed computing systems. We are pursuing the formal specification and verification of system components (e.g. hardware, assemblers, compilers, operating systems, etc.) as well as the *stacking* of these proved components. The intention is to create a hierarchical stack of verified components that build on top of one another so that eventually we can verify a program in a high level language and know that the compiler, assembler, run-time system, and even the hardware have all been formally verified to operate to specification. This effort has the long-term goal of demonstrating a methodology for the development of computing systems with reliability that far exceeds today's standards.

To focus our work we have been working on the verification of what we call a *short-stack*. This stack embodies the ideas of system verification, but at a manageable size. We have completed a short stack consisting of the micro-Gypsy compiler, the Piton assembler, and the FM8502 microprocessor. The verified compiler emits Piton assembly code. Piton assembles to FM8502 machine code. FM8502 is implemented in terms of a gate level description. This is a major milestone. We plan to add some type of I/O to this stack, which will undoubtedly raise numerous interesting problems.

Trusted Systems is also supporting the formal specification of the semantics of a subset of the DoD standard programming language Ada. The Ada work is a step towards the provision of mechanized support for formal reasoning about Ada programs. The driving force behind this effort is the recognition of the tremendous near term practical utility of such a capability.

3.1 Hardware Verification

Several interrelated hardware specification and verification efforts are underway. Hardware provides the foundation for all computing systems. Without correct, reliable hardware no amount of software verification can provide trustworthy systems.

Our hardware effort contains several thrusts: formal modeling, verification techniques, device/system specification and verification, and test-vector generation. Our modeling effort is concerned with the construction of formal models for hardware devices. This requires both an understanding of hardware device properties and of formal systems. Our modeling effort provides our hardware foundation. To verify hardware, we have investigated various verification techniques, such as exhaustive testing, decision procedures, etc.; however, our most successful technique involves an axiomatic approach. We have been able to specify and verify a microprocessor of a complexity similar to a PDP-11 insofar as arithmetic operations are concerned. We have expanded this work in the specification and verification of a Universal Asynchronous Receiver/Transmitter (UART). The specification of a UART involves I/O outside the state-space of the UART, thus requiring the specification of the external message (bit) streams. Because UART's provide a model for many types of I/O, we continue to investigate better methods of specification and verification. Test-vector generation has previously been a monolithic effort where test-vector generation could only be accomplished by considering an entire device. We have constructed a compositional test-vector generation method that allows tests for subparts to be generated first and then combined to provide test vectors for an entire circuit.

3.1-A Hardware Models

We are attempting to construct a new hardware model. Previously, we used functions in the Boyer-Moore logic to represent specifications for *n*-bit sized hardware. We then expanded these functions into gate-trees to obtain actual circuits. This approach suffers from an inability to determine before expansion whether a specific hardware function will expand into hardware primitives only. In addition, there was no formal definition for what was "good" hardware. This problem is compounded by the use of recursive functions which were supposed to model synchronous circuits. A new hardware model is being designed to make clear what we consider "good" hardware. Recursive functions are used to create gate-graphs (e.g., fan-out can be specified) directly, instead of having to be introduced by the expansion process.

Our new hardware model will provide new gate types and allow analysis to be performed directly on circuit formulas. We believe our new model will be detailed enough for an attempt at building real, functioning hardware devices from our formal hardware descriptions. This is an important area of research we need to explore more fully. We have demonstrated the ability to verify large hardware devices constructed from simple digital logic gates; however, we must be able to turn these logically perfect (gate-graphs) designs into actual hardware with complete confidence.

Another important issue of constructing formal hardware models within formal theories is the difficulty of proofs about hardware circuits within the model. If the model is simple, then proofs are easier; and if the model is complex, proofs are more difficult. It is important when constructing hardware models to consider the effect these models have on our ability to perform proofs. Indeed, since we employ mechanical theorem provers to perform our proofs, it is necessary to engineer the models very carefully to be able to take advantage of our mechanical theorem proving capabilities. As we develop our models, we continually check our ability to prove the correctness of hardware circuits.

3.1-B Core MIPS

We have employed our specification techniques to fully specify Core MIPS, excluding floating-point. Core MIPS is a DARPA-funded, natural language specification of an assembly language which will be shared among the family of MIPS processors. Our Core MIPS processor specification closely follows the DARPA description [Firth87], and is similar to Hunt's specification for the FM8501 [Hunt87]. The specification is cast as an instruction interpreter; that is, a precise specification of every machine instruction is presented by showing its effect on the state of a MIPS machine.

3.1-C UART

The attempt to develop a formally specified UART is focusing our attention on the specification of I/O. Functions are fine for specifying devices in which their next state is a function of their current state. However, to specify devices which communicate with other devices, it is simpler to constrain (or specify) the history of their communication ports with predicates. This technique, although more general than our previous functional approach, has several difficulties which we are exploring: predicates are difficult to execute and we have little experience doing predicate style proofs. Another problem with a predicate approach is the potential for the construction of predicates which are not satisfiable. Unless witness functions can be constructed, it is not possible to execute predicates; therefore, simulation is not possible. Even considering these problems, predicates offer a much more elegant way of specifying streams of data and their interconnection.

We are employing our predicate approach in the specification and verification of a UART. Using predicates, a simple parallel to serial converter which "transmits" bytes in a standard format (1 start bit, 8 data bits, 1 stop bit) has been specified. This device is equivalent to approximately 90 gates. We intend to extend our present method to specify and verify a more general UART containing realistic time and signal constraints.

3.1-D Test Vector Generation

An algorithm for compositional test vector generation has been completed [Vose88]. Normally, each time a digital hardware circuit is changed, however slightly, a new set of test vectors must be generated by reconsidering the entire circuit. Using this new compositional approach allows more rapid test vector generation for small perturbations in designs. This test vector generation method uses formal (FM8501-like) circuit specifications as input, allowing generation of test vectors for verified circuit designs.

3.2 The Piton Assembler

To verify a system one must be able to *stack* verified components. That is, one must be able to use what was proved about one level to construct the proof of the next.

The Piton project is the first instance of stacking two verified components. In 1985 Warren Hunt designed, specified and proved the correctness of the FM8501 microprocessor. The FM8501 is a 16-bit, 8 register general purpose processor implementing a machine language with a conventional orthogonal instruction set. Hunt formally described the instruction set of the FM8501 by defining an interpreter for it in Boyer-Moore Logic. Hunt also formally described the combinational logic and a register-transfer model that he claimed implemented the instruction set. The Boyer-Moore theorem prover was then used to prove that the register-transfer model correctly implemented the machine code interpreter.

The existence of a verified design for a general-purpose processor clearly suggested the idea of using that processor as the *delivery vehicle* for some *trusted system* such as an encryption box, a verifiable high-level language, or perhaps even a program verification system. The first step was to design, implement and verify an assembly-level language on FM8501. We named the language *Piton* after the spikes driven into rocks by mountain climbers to secure their ropes.

When the Piton project began, the intended hardware base was FM8501. Early in the project we requested two changes to FM8501, which were implemented and verified by Warren Hunt. The modified machine was called the *FM8502*. The changes were (a) an increase in the word width from 16 to 32 bits and (b) the allocation of additional bits in the instruction word format to permit individual control over which of the alu condition code flags were stored. Because of the nature of the original FM8501 design, and the specification style, these changes were easy to make and to verify. Indeed, the FM8502 proof was produced from the FM8501 script without human assistance.

We wanted Piton's semantics to be relatively clean so that it was straightforward to prove properties of Piton programs. But the cleanliness could not come by making unrealistic assumptions, since it was our goal to implement the language and prove the implementation correct. Finally, we wanted the implementation to be efficient so that one could actually use Piton in verified applications. We imagine using Piton directly to write encryption programs and indirectly as the target language for verified high-level language compilers.

Among the features provided by Piton are:

- execute-only program space
- named read/write global data spaces randomly accessed as one-dimensional arrays
- recursive subroutine call and return
- provision of named formal parameters and stack-based parameter passing
- provision of named temporary variables allocated and initialized to constants on call
- a user-visible temporary stack
- seven abstract data types: integers, natural numbers, bit vectors, Booleans, data addresses, program addresses (labels), and subroutine names.
- stack-based instructions for manipulating the various abstract objects
- standard flow-of-control instructions
- instructions for determining resource limitations

Among the significant achievements of the Piton project are the following.

Piton truly provides abstract objects and a new programming language on top of a much lower level machine. Much has been written about this classic problem, but previous attempts to deal with it formally and mechanically have been unsatisfactory. We have in mind specifically the work related to the SRI Hierarchical Design Methodology [Robinson&Levitt77] and its use in the Provably Secure Operating System (PSOS) [Neumann75] and the Software Implemented Fault Tolerant (SIFT) operating system [Melliar-Smith&Schwartz81, Stanat84]. While virtually all of the issues are correctly intuited, we personally find great joy in seeing their formalization.

The commitment to stacking has had several effects. The desire to implement Piton forced into its design such practical considerations as the finite resources of the host machine. The desire to use Piton forced us to reflect the resource limits into the language itself. Programs that cannot anticipate the imminent exhaustion of their resources are impractical.

3.3 Compiler Verification

The verification of a compiler for micro-Gypsy [Young86] is complete. micro-Gypsy is a subset of the Gypsy language which includes exceptions, records, control flow statements, and procedure calls. The formal semantics of micro-Gypsy are defined in Boyer-Moore Logic. The compiler generates Piton assembly code and allows the execution of verified micro-Gypsy programs on a completely verified system. What is proven is that the Piton emitted by the compiler executes in conformance with the semantic definition of micro-Gypsy.

3.4 Ada Verification

We are attempting to develop an effective and sound logical basis for reasoning about AVA (Annotated Verifiable Ada) programs. AVA is a formally defined subset of the Ada programming language [DoD83]. In developing this definition, we are following an approach similar to that used to define micro-Gypsy. See [Young86]. The measure of adequacy of the subset is that it be sufficient to implement a message flow modulator (MFM), an application that has been specified, implemented, and proved correct in Gypsy [Good82b].

A small subset of AVA, called *nano-AVA*, has been used to refine our basic approach to the definition [Smith88]. We are now considering *micro-AVA*, a slightly extended nano-AVA.

At the outset, we admitted the extreme possibility that there exists no verifiable subset of Ada. However, we currently believe that there is a verifiable Ada subset that contains at least something comparable to micro-Gypsy and that AVA can be defined with the same basic approach used to define micro-Gypsy. This definitional effort provides a number of important benefits.

1. It provides the logical basis for an eventual production quality AVA verification system.
2. It provides the base for an enlarged verifiable subset of Ada.
3. If the micro-Gypsy definition approach can be followed closely, the AVA definition provides a basis for a proved AVA compiler and a proved run-time executive that are both targeted for proved hardware (as with micro-Gypsy). This offers the possibility of completely proved AVA systems.

3.4-A nano- and micro-AVA

Nano-AVA has been defined using a denotational approach. This definition was translated into Boyer-Moore Logic and some simple proofs about a trivial swap program were performed.

The micro-AVA Reference Manual exists in draft form. The nano-AVA denotational definition has been revised to support the extension of the definition further towards full AVA. Both of these efforts have turned up further problems with the Ada Reference Manual. For example, the visibility rules in 8.3 seem to assert that no object declaration is legal. A number of such problems have been encountered and for many cases there seem to be obvious fixes; however, in some cases it is not at all clear what is intended.

3.4-B Message Flow Modulator

A third revision of the message flow modulator (MFM) has been completed. The MFM reads in messages over an input channel and checks them for the presence of strings matching a set of prestored patterns. If no matches are found the message is passed to the output channel. Otherwise the message and the offending patterns are logged.

This MFM was believed to completely satisfy the original Gypsy specification. It has successfully passed the version of the Gypsy test suite that we have been able to resurrect and additional tests developed to check some properties untested by that suite. During review at an AVA group meeting one potential bug was detected in the pattern admissability test. A class of patterns were admitted that should be ruled out. The addition of a trivial test has corrected this. The MFM has been partially specified in Boyer-Moore logic in order to get a feeling for the kinds of annotations that will ultimately be useful in AVA.

The MFM was developed, compiled and run on a Sun 3/280 using the Verdix Ada compiler. The program was not written with any intention of being portable. But, out of curiousity, we also tested it on a Sun using the Telesoft compiler. This test yielded identical results to the Verdix run. Then we tried using the DEC compiler on a VAX. As we expected, the execution failed because of the different handling of line-feeds on the two machines. Had we been intending portability from the start, this particular failure would not have arisen. (And in fact could be trivially remedied.)

4. Gypsy Verification Environment

The Gypsy methodology is an integrated system of methods, languages, and tools for designing and building formally verified software systems. The methods provide for the specification and coding of systems that can be rigorously verified by logical deduction to always run according to specification. The methodology makes use of the Gypsy Verification Environment (GVE) to provide automated support. The GVE is a large interactive system that maintains a Gypsy program description library and provides a highly integrated set of tools for implementing the specification, programming, and verification methods. The GVE is one of two systems designated by the National Computer Security Center as applicable to the formal specification and verification of the security properties of computer software systems.

The Gypsy Methodology has been used successively in several substantial applications. These include message switching systems [Good82c], communication protocols [DiVito82], security kernels, monitoring of inter-process communication [EPI82], and secure networks (Ford Multinet).

We continue to maintain and enhance the Gypsy Verification Environment. We are also providing assistance in the application of the GVE to particular problems.

Major work on the Gypsy Verification Environment (aside from normal maintenance activities) has consisted of efforts to fill in known holes in the support that the system provides to users. The primary areas of concern have been the well-formedness of expressions in the theorem prover and the proper tracking of dependencies between units in a user's database. The first of these has been corrected. The second is in progress and should be complete in October. In addition, the GVE has been ported to Sun workstations and is currently undergoing test prior to release. We hope the increased availability of the GVE due to this port will encourage broader experimentation with the GVE.

1. Well Formedness - In the past there was nothing to prevent a user from introducing various sorts of ill-formed expressions into the theorem prover. The prover assumes that everything that it sees is well-formed. This allowed users to take advantage of ill-founded recursions in function definitions (f(x) = not f(x)) and undefined primitive operations (x div 0 = y) to prove false theorems. Operations of this sort are now prevented.

2. Incremental Development - The GVE tracks dependencies between various Gypsy units in the user's database. Thus, when you change the exit specification of procedure P which is called by procedure R the system should indicate that you need to regenerate the verification conditions of R. There were untracked dependencies in the old system (primarily missing information indicating the dependence of a proof on a unit whose definition was imported in the course of the proof). Work is in progress (to be completed in October) to redo this entire mechanism to make it sound.

5. How Can This Technology Be Applied Currently?

We need examples of mathematical proofs of program properties (not just security properties). We need a larger body of trained individuals who can apply formal methods to the specification and proof of software. The only way to accomplish this is to apply the existing technology to selected problems.

It is possible, with currently available tools and/or pencil and paper, to write formal specifications for components of the SDS software. These specifications can be used to produce proofs about the behavior of the software. If a common specification language can be pinned down it is my impression that these specifications might be useful for some of the testing methodologies described in *SDS Software Testing and Evaluation*.

It is important to select such applications carefully. They should be both significant components of SDS and at the same time tractable in terms of specification and proof. It would be extremely useful to focus a substantial portion of such an effort at a very low level, say at the assembly code level. This would require a fairly substantial effort to specify the semantics of some particular assembly language.

Currently applicable tools include the Boyer-Moore Theorem Prover, the Gypsy Verification Environment, Eves [Craigen86], EHDM [Crow85a, Crow85b], plus an assortment of approaches being developed in Europe (for an overview see [Lindsay88]). Some of these systems require more training than others. Normally with more training you get an enhanced proof capability. For a new user with a programming background, I am fairly sure the Boyer-Moore Theorem Prover is the most difficult to use of the listed technologies. On the other hand it has been used to prove substantially more difficult systems than the other technologies.

6. How Can This Application Be Accelerated?

In the long term, a vast amount of work needs to be done.

Below are some miscellaneous, fairly near term projects which would seem to provide some payoff.

6.1 Real Time

A *functional property* of a program is one involving some predicate on the value of the result produced. A *real-time property* of a program states a fact about the time at which a result is produced. Research in the verification of real-time properties of programs is in its infancy. Some work has been done to address the problem of how to specify and design real-time systems, but this work will likely lead to a difficulty faced by current technology in the verification of functional properties of programs: The behavior of a "verified" program as predicted by its correctness proof may not correspond to the behavior of the program when compiled and run on a real system. There is a large gap between design proofs and proofs of actual running code.

We want to investigate the proofs of real-time properties of programs starting on the ground floor, where the ground floor is the specification of the architecture level of a hardware processor. At this level, knowledge of the execution times of instructions and the arrival of interrupts is available. Given a formal description of a processor, we would like to investigate what kinds of real-time properties can be proved of running machine code. There are a number of directions which this may take. Examples are the behavior of a simple communications link, and the real-time behavior of tasks scheduled by an operating system kernel.

The long-range goal of producing running code whose real-time properties are verified involves the following problems.

1. How to write a specification for a processor which is both adequate for proving real-time properties, and which can be implemented in hardware.
2. How to go about specifying real-time properties of programs in such a way that machine code can be verified.

6.2 Floating Point

Much of the SDS software will depend on floating point computations. In order to reason about these systems it would be useful to have at hand a library of proved properties of the IEEE floating point standard. Even better would be to demonstrate that some particular piece of hardware is implemented precisely in accordance with a formal definition of the standard. Thus three interesting pieces of work include:

1. Write a formal version of the IEEE Floating Point Standard.
2. Prove some useful properties of the standard based on this formal definition.
3. Prove that a description of a floating point processor at the level of hardware gates implements the standard.

We already have the technology to carry out this project.

6.3 Ada Language Definition

Any modifications to the Ada definition should be sensitive to formal requirements. Or at least allow them! For example something like a **pragma verifiable** could be added to the language. If a compiler honored this pragma it would guarantee not to use the optimization latitude provided by section 11.6 of the Ada Reference Manual. It might honor left to right order of evaluation and pass by reference. The intent of such a pragma (there probably should be multiple ones, e.g. **pragma left_to_right**, **pragma pass_by_reference**, **pragma time_slice_scheduling**, etc.) would be to guarantee the programmer that he will at least be alerted when a particular compiler failed to satisfy his expectations. (This would also require the standard be modified to insist that a compiler notify the user when it failed to recognize an implementation-defined pragma.)

The major difficulty with the current Ada standard is that it is impossible to formalize because it is internally inconsistent. That aside, definition is complicated by the explicit admission of multiple behaviors as part of the standard. Nonetheless, formally based tools can make an engineering contribution to enhancing the reliability of Ada programs. They can be based on formal definitions of Ada subsets that eliminate inconsistencies and select particular behaviors. Such choices can be made explicit. This would allow users to know the extent to which predictions made by the formal tools can be depended on when the Ada is compiled with a particular compiler, assuming that one can determine what choices the compiler implementors adopted. In an engineering sense, the application of formal tools will enhance the reliability of the software, even it it does not provide the mathematically absolute predictions that we would prefer.

6.4 Mathematical Spread Sheets

A spread sheet allows the incremental construction of a set of simple relations that compute a complex result. This sort of interface is needed by the theorem proving technology. At the moment these interfaces seem to be limited to Emacs [Stallman87].

There is a language problem as well as an interface problem. Most non-Lisp programmers seem more at home with the syntax of Pascal-like languages than with Boyer-Moore Logic. Mathematicians would prefer that the tools provide direct support for mathematical notations, perhaps by making use of Postscript windows.

6.5 Hardware

The primary impact that hardware verification might have on SDS is in providing guidance in the specification of hardware devices such that software implemented on them can be verified.

In addition to more work on the specification of different kinds of hardware components it is important to continue moving in the direction of fabrication, dealing with the problems that arise as we get closer to the physical world.

References

[Bevier87a] William R. Bevier.
A Verified Operating System Kernel.

Technical Report CLI-11, Computational Logic, Inc., October 1987.
Ph.D. thesis, The University of Texas at Austin.

[Bevier87b] William R. Bevier.
The Formal Specification and Definition of KIT.
Technical Report CLI-16, Computational Logic, Inc., December 1987.

[Bevier88] Bill Bevier.
A Library for Hardware Verification.
Technical Report Internal Note 57, Computational Logic, Inc., June 1988.
Draft.

[Boyer&Moore79] R.S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer&Moore81] R.S. Boyer and J S. Moore.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
In R.S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[Boyer&Moore83] Robert S. Boyer and J Strother Moore.
A Mechanical Proof of the Turing Completeness of Pure Lisp.
Technical Report ICSCA-CMP-37, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1983.

[Boyer&Moore84] R.S. Boyer and J S. Moore.
Proof Checking the RSA Public Key Encryption Algorithm.
American Mathematical Monthly 91(3):181-189, 1984.

[Boyer&Moore88] R.S. Boyer and J S. Moore.
A User's Manual for a Computational Logic.
Technical Report CLI-18, Computational Logic, Inc., 1988.

[Craigen86] Dan Craigen.
A Description of m-Verdi [Working Draft]
I.P. Sharp Associates, Ltd., 1986.

[Crow85a] Judith Crow, Dorothy Denning, Peter Ladkin, Michael Melliar-Smith, John Rushby, Richard Schwartz, Robert Shostak, Friedrich von Henke.
SRI Verification System Version 1.8 Specification Language Description.
SRI International Computer Science Laboratory, 1985.

[Crow85b] Judith Crow, Dorothy Denning, Peter Ladkin, Michael Melliar-Smith, John Rushby, Richard Schwartz, Robert Shostak, Friedrich von Henke.
SRI Verification System Version 1.8 User's Guide.
SRI International Computer Science Laboratory, 1985.

[Dijkstra68] E.W. Dijkstra.
A Constructive Approach to the Problem of Program Correctness.
BIT 8-3, 1968.

[DiVito82] B.L. DiVito.
Verification of Communications Protocols and Abstract Process Models.
Technical Report ICSCA-CMP-25, University of Texas at Austin, 1982.

[DoD83] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1983.
ANSI/MIL-STD-1815 A.

[EPI82] Formal Verification of a Communications Processor.
Final Report, Contract MDA 904-80-C-0481, Institute for Computing Science, The University of Texas at Austin.
February, 1982.

[Firth87] Robert Firth.
Core Set of Assembly Language Instructions for MIPS-based Microprocessors.
January, 1987.

[Good82a] Donald I. Good.
The Proof of a Distributed System in Gypsy.
In *Formal Specification - Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar - M. J. Elphick (Ed.)*. September, 1982.
Also Technical Report #30, Institute for Computing Science, The University of Texas at Austin.

[Good82b] Donald I. Good, Ann E. Siebert, Lawrence M. Smith.
OSIS Message Flow Modulator - Status Report.
Internal Note #36A, Institute for Computing Science, The University of Texas at Austin.
April, 1982.

[Good82c] Donald I. Good, Ann .E. Siebert, Lawrence M. Smith.
Message Flow Modulator - Final Report.
Technical Report CMP-34, Institute for Computing Science, University of Texas at Austin, December 1982.

[Good88a] Donald I. Good.
Predicting Computer Behavior.
Technical Report CLI-20, Computational Logic, Inc., May, 1988.

[Good88b] Michael K. Smith, Donald I. Good, Benedetto L. DiVito.
Using the Gypsy Methodology
Computational Logic Inc., 1988.
Revised January 1988.

[Hoare82] C. A. R. Hoare.
Programming is an Engineering Profession.
Technical Report PRG-27, Programming Research Group, Oxford University Computing Laboratory, 1982.

[Hunt85] Warren A. Hunt, Jr.
FM8501: A Verified Microprocessor.
Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[Hunt87] Warren A. Hunt, Jr.
The Mechanical Verification of a Microprocessor Design.
Technical Report CLI-6, Computational Logic, Inc., 1987.

[Lindsay88] Peter A. Lindsay.
A survey of mechanical support for formal reasoning.
Software Engineering Journal, January, 1988.

[Melliar-Smith&Schwartz81]
P.M. Melliar-Smith and R. Schwartz.
Hierarchical Specification of the SIFT Fault-Tolerant Flight Control System.
Technical Report CSL-123, Computer Science Lab, SRI International, Menlo Park, CA, 1981.

[Misra81] J. Misra and K. M. Chandy.
Proofs of Networks of Processes.
IEEE Transactions on Software Engineering SE-7(4), July, 1981.

[Moore88] J Strother Moore.
PITON: A Verified Assembly Level Language.
Technical Report CLI-22, Computational Logic, Inc., 1988.

[Neumann75] P.G. Neumann, L. Robinson, K.N. Levitt, R.S. Boyer, A.R. Saxena.
A Provably Secure Operating System.
In *USAECOM*. SRI, June, 1975.
SRI Project 2581.

[Robinson&Levitt77]
L. Robinson and K. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Communications of the ACM 20(4), April, 1977.

[Russinoff83] David M. Russinoff.
An Experiment with the Boyer-Moore Program Verification System: A Proof of Wilson's Theorem.
Masters Thesis, Department of Computer Sciences, University of Texas at Austin, 1983.

[Shankar87] N. Shankar.
Proof Checking Metamathematics: Volumes I and II.
Technical Report CLI-9, Computational Logic, Inc., April, 1987.

[Smith88] Michael K. Smith, Dan Craigen, and Mark Saaltink.
The nanoAVA Definition.
Technical Report CLI-21, Computational Logic, Inc., May, 1988.
Draft.

[Stallman87] Richard M. Stallman.
GNU EMACS Manual
Sixth edition, Free Software Foundation, 1987.

[Stanat84] D.F. Stanat, T.A. Thomas and J.R. Dunham.
Proceedings of a Formal Verification/Design Proof Peer Review.
Technical Report RTI/2094/13-01F, Research Triangle Institute, 1984.

[Vose88] Michael D. Vose.
Applications of Compositional Test Generation to Recursively Specify Combinational Logic.
Technical Report CLI-26, Computational Logic, Inc., August, 1988.

[Young86] William D. Young.
A Verified Compiler for Micro Gypsy.
PhD thesis, The University of Texas at Austin, In Progress 1986.

Panel: Measurement

C.J. Linn, Chair

Panelists: V.R. Basili
M. Evangelist
H.D. Rombach
R.W. Selby
V.Y. Shen

SDIO Workshop Position Paper

Victor R. Basili
Department of Computer Science
University of Maryland

I. Critical Gaps

A. Evaluation Technology

There is a great deal of work that needs to be done in developing mechanisms for evaluating software.

First, we need to define the qualities we are looking for. For example are we evaluating with respect to reliability, performance, functionality, correctness, etc.? We need clear, operational definitions of these terms, that allow us to measure if we have achieved them, and to what level. We need to have models of the various products and their attributes.

Second, we need to have models of the various processes that create the products on the basis that there is a direct correlation between the process and product attributes. We need to identify them, to determine if the process is being performed right. This requires that we characterize and measure the software development processes.

Development methods are often heuristic and not formal. They require interpretation and evaluation as to whether they are being performed appropriately, if at all. Our experience in trying to characterize methods so that they can be evaluated as to whether they are being applied correctly demonstrates the lack of precision in the specification of the methods. In a study of the state of the practice in industry, we found that very few organizations are using the methods and tools appropriately. This is largely due to the heuristic nature of many of the methods, i.e. if one developer "does it right" it is hard to explain to another developer exactly what he/she is doing.

There needs to be a mechanism for evaluating how well the process is being performed so that it can be improved. In our studies we found very few organizations running post mortems that would allow them to learn how to better develop software.

Third, these models and processes need to be tailored to the specific domain in which we are working, e.g. in our case to SDS software. All software environments are different and models and processes that work in one environment do not necessarily work in another. In the SEL, we began by applying other people's models and metrics to see what occurred. Usually, the model or the metric applied didn't explain what was really happening. Those models and metrics were developed for specific environmental characteristics which did not hold in our environment. However, in understanding why the model or metric didn't work, we learned a great deal. We need to learn how to tailor processes and products without loosing reuse.

Fourth, we need to identify metrics based upon those models, not metrics developed in isolation. We have found many organizations measuring software projects, but most of these measurement programs were useless. Their measurements are bottom-up based blindly upon models and metrics in the literature, rather than top down based on an understanding of their own processes, products, and environment. (We use the goal/question/metric paradigm for this.) They have not initiated the first essential step toward quality assessment and control: quantitatively characterizing their environments to better understand the nature of their business.

We learned that we needed to quantitatively characterize the local software development process and product to better understand it. (It is amazing how much people know but don't tell you because they don't realize it is important or don't have a forum for expressing it.) We encode this information in subjective as well as objective metrics. Subjective metrics capture knowledge of the environment that exists in people's heads, but can't be easily quantified e.g. the extent to which a particular method is taught, understood or applied. This information can be categorized on a quantitative scale to a reasonable degree of accuracy.

Fifth, we need to develop experimental techniques that can be used in conjunction with software development and maintenance. Clearly the domain in which we must experiment and learn is the SDIO development environment.

The development and maintenance environments must be prepared for measurement and evaluation. There is a planning phase necessary for this activity and the activity must be carefully embedded in the process. This planning phase must take into account the experimental design appropriate for the situation.

It is necessary to decide what we want to measure, how we are going to measure it and how we are going to interpret the results. Part of the planning process deals with choosing the appropriate set of measures, not too many or too few, evaluating the cost of collection and analysis, and determining how it will be used. Often data is collected but not used because it was not appropriately planned for.

Sixth, we need to develop technologies that will help us interpret the data from those experiments based on the models. This involves the development of data bases of information that provide baselines with regard to the metrics and allow us to deal with statistical quality control for software.

Seventh, we need to integrate with data base and AI technologies to build information bases. The storage of information and the development of measurement systems require that we have large experience bases of information that contain the information necessary to characterize, evaluate, and learn.

Eight, we need to develop process models that support the learning and feedback process so we can make improvements in the software technology and methods. Measurement must be an active process in which we learn how to better do things. A process model must support this.

In most organizations, there is not the appropriate organizational level commitment needed to gather characterizing data across the organization. (How do you decide if you've gotten better if you don't know where you are?) There is not proper analysis of metrics across similar projects. There is not learning from the measurement process about how to better build products, use better methods, and do better measurement. All of these steps take careful planning and organization.

B. Validation and Verification Technology

There is a lack of understanding in the software process with regard to levels of technologies. For example, assume the following model and set of definitions. Define a technique as the basic mechanism for constructing or assessing the software, e.g. reading. Define a method as an organized approach based upon applying some technique, e.g. design inspections. Define a process as an integrated set of methods that covers the life cycle, e.g. an iterative enhancement model using structured design, design inspections, etc. Define an engineering process as the application and tailoring of activities to the problem, project and organization.

Then there are several areas where there are gaps in the technology.

First, reading has not received sufficient attention as a technology. There is little training in the technology, little development in alternate forms, e.g. reading sequentially vs. stepwise abstraction vs. path following. Little emphasis on perspective, e.g. how does one read from the perspective of the customer, tester, etc.

Second, there are areas where methods have not been well defined. For example, testing methods. Most of the testing approaches are techniques rather than methods. Where are the organization mechanisms for testing like design inspections for reading? What few that do exist have not been institutionalized very well.

Third, most validation and verification techniques and methods need to be engineered. Where are the error, fault and failure histories for similar applications and similar constructive processes when it comes to tailoring reading and testing techniques.

Fourth, there is a lack of traceability of the techniques through the life cycle. How do requirements flow into design into code and how do the reading and testing processes integrate to capture that traceability and complement one another.

II. Potential Achievements

In the near term, we can set goals for the SDS software, build preliminary models of the processes and products, pick a measurement methodology program, build an experimental laboratory consisting of data collected from the SDIO software being built. Models for such programs exist, e.g. the SEL, and need to be tailored for this organization.

We can also work on characterizing the methods and tools to be used and that will help us in choosing the appropriate ones for SDS. We can perform some early experiments with these methods and tools in the actual environment in which they are developed in order to evaluate their effects.

We can begin to create a data base of information on such things as errors, faults and failures and resource usage based upon the models developed.

In the long term, we can create an experience base as we learn more about how to take advantage of the data base and AI technologies. We can modify and refine our methods and goals based upon our experiences and the data collected.

III. Current Status of Work

1. SEL

The SEL has existed for over a dozen years and has been the major mechanism for experimentation, evaluation and learning about software processes and products. Many of the problems expressed in the first section of this report came from studies within the SEL. There are several studies currently underway in the SEL.

a. The Cleanroom Experiment

The Cleanroom approach is based upon the work of Harlan Mills at IBM. We had performed early controlled experiments in the approach that showed it effective in a small controlled environment. We are now applying it to software being developed at NASA. The experimental model is a case study and goals, questions and metrics have been developed for the study.

b. Comparison of Ada and FORTRAN Developments

This is the development of two implementations of the same system, one in FORTRAN and one in Ada. We measured both product and process aspects of the developments to try to understand the effect of the change for NASA to Ada. We have written lessons learned documents for each of the phases of the development in order to pass the information learned on to future projects.

We are currently analyzing the data in greater detail in order to answer questions such as: what are the differences in errors, faults and failures? what are the costs by class of each? where were resources expended differently and how could they be minimized?

c. Study of Reuse

The reuse project is an attempt to build models of reuse and discover the cost of reuse for both the FORTRAN and the Ada environments.

d. Study of Maintenance

This project takes the work we have done for the development environment and tries to extend it to the maintenance environment.

2. The TAME Project

The TAME Project is based on the idea that measurement, using operationally defined goals, can provide the mechanism needed for tailoring and feedback. Because the measurement process is goal oriented, it must be top down, with operational goals defining the metrics needed to evaluate the project's successes and failures, permitting interpretation in a specific context, and providing feedback for improving the process and the product from the perspective of the specific project, the overall environment and the organization.

Based upon our experiences in trying to evaluate and improve the quality in several organizations, we have learned that the development of a measurement and analysis program that extends through the entire life cycle is a necessity. Such a program requires an improvement-oriented life cycle model that has four essential aspects:

1. Planning. There are three integrated activities to planning that are iteratively applied:

(a) Characterizing the environment. This involves data that characterizes the resource usage, change and defect histories, product dimensions and environmental aspects for prior projects and predictions for the current project. It provides a quantitative analysis of the environment and a model of the project in the context of that environment.

(b) Defining quality operationally relative to the customer, project, and organization. This consists of a top-down analysis of goals that iteratively decomposes high-level goals into detailed sub-goals. The iteration terminates when it has produced sub-goals that we can measure directly. This approach differs from the usual in that it defines quality goals relative to a specific project and from several perspectives. The customer, the developer, and the development manager all contribute to goal definition. It is, however, the explicit linkage between goals and measurement that distinguishes this approach. This not only defines what good is but provides a focus for what metrics are needed.

(c) Choosing and tailoring the process model, methods, and tools to satisfy the project goals relative to the characterized environment. Understanding the environment quantitatively allows us to choose the appropriate process model and fine tune the methods and tools needed to be most effective. For example, knowing prior defect histories allows us to choose and fine tune the appropriate constructive methods for preventing those defects during development (e.g. training in the application to prevent errors in the problem statement) and assessment methods that have been historically most effective in detecting those defects (e.g. reading by step-wise abstraction for interface faults).

2. Analysis. We must conduct data analysis during and after the project. The information should be disseminated to the responsible organizations. The operational definitions of quality provide traceability from goals to metrics and back. This permits the measurement to be interpreted in context ensuring a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis needed.

3. Learning and Feedback. The results of the analysis and interpretation phase can be fed back to the organization to change the way it does business based upon explicitly determined successes and failures. For example, understanding that we are allowing faults of omission to pass through the inspection process and be caught in system test provides explicit information on how we should modify the inspection process. Quantitative histories can improve that process. In this way, hard-won experience is propagated throughout the organization. We can learn how to improve quality and productivity, and how to improve definition and assessment of goals. This step involves the organization of the encoded knowledge into an information repository to help improve planning, development, and assessment.

We have recently begun a project which will automate as much of this process as possible. This project is call the TAME system (Tailoring a Measurement Environment).

3. The TAME System

The TAME system provides a mechanism for managers and engineers to develop project specific goals, and generate operational definitions based upon these goals that specify the appropriate metrics needed for evaluation. The evaluation and feedback can be done in real time as well as help prepare for post mortems. It will help in the tailoring of the software development process. It consists of four major components: user interface, an evaluation mechanism, a measurement mechanism and an information base. The user interface provides the support needed for helping managers develop operational goals, the evaluation mechanism provides feedback based upon the various project goals, the measurement mechanism provides whatever measurement capabilities are needed for automated collection of metrics, the information base contains the historical data base, all project documents, the current project data base, a goal/question/metric data base and any other information necessary for support.

The short range (1-3 years) goal for the TAME system is to build the evaluation environment. The mid-range goal (3-5 years) is to integrate the system into one or more existing or future development or maintenance environments. The long range goal (5-8 years) is to tailor those environments for specific organizations and projects.

The TAME system is an ambitious project. It is assumed it will evolve over time and that we will learn a great deal from formalizing the various aspects of the TAME project as well as integrating the various paradigms.

4. Testing Methodology

This project, done under the improvement process model and the goal question metric paradigm is attempting to build a method for applying testing technologies. The basic approach is to take customer needs, trace them into system requirements, through design, to code using such matrices as (customer need, requirements), (requirements, design components), and (design components, code modules). It then takes a requirements based testing technique and evaluates it based upon requirements, design and code coverage (in the latter case using structural coverage metrics). A pilot study of the application of the technology is being organized.

Dr. Michael Evangelist
MCC
3500 W. Balcones Center Dr.
Austin, TX 78759

Even without considerable experience in the "upstream" problems of software development, one must be greatly disturbed by the deficiencies of current T&E technologies. [There is], in fact, compelling evidence that few sophisticated technologies and methods will be available to support SDS implementation, testing, and maintenance in the visible future. The testing community has not been successful in applying their simple theoretical results to large, realistic systems; verification has worked only on small (sequential) programs, and is very expensive; and software metrics research has failed to produce measures more useful than LOC.

I will say little about evaluation research; my [published] papers give my pessimistic appraisal of the area. Software measurement will not be useful until researchers abandon the a priori approach (which has resulted in such measures as cyclomatic complexity and the Halstead metrics) and concentrate on developing customized measures of properties that are strongly related to explicit project goals. Once such a foundation is built, data will be available from which analytical metrics can be derived. That is, the process will be much like the relationship between observation and theory in the physical sciences. Metrics research has too frequently used poorly justified theory that has not been driven by characteristics of real systems. To take one prominent example: McCabe advocated cyclomatic complexity as a measure of "testability," but he gave no evidence that this measure was related to the effort put into testing for any realistic system. (I countered his analytical argument in [Evan84].) Unfortunately, Basili's group at the University of Maryland is the only one I know of taking the goals-related approach.

The state-of-the-art is embarrassingly deficient, primarily because 1) it is not informed by experience from traditional engineering fields; and 2) it is not tied to the needs of software developers and testers.

From traditional fields we should understand the need for engineering standards for re-using designs and implementations. In so doing, we will gain not merely an increase in productivity, but, more importantly here, verifiability. Bridge builders, for example, do not re-prove the viability of a design for each new bridge. Unfortunately, current research in verification and testing encourages that approach. T&E tools and methods must not be left to the intuitions of testing researchers who have never tested a large system themselves and who, therefore, make no attempt to support existing successful practice.

We cannot depend on new results in T&E to help in the near term with building these large, complex systems. The DoD should support promising T&E research, but the major focus must be on avoiding errors in the requirements, design, and coding phases. To do that, we need a major research initiative in these areas, particularly in executable design languages and methods and the tools to support them. The deficiencies in code-level testing become less important when we raise the quality of designs. Once we develop these technologies, we will also have a firmer foundation for testing and evaluation at the design level and for tracing those results to the implementation.

To my mind, code verification is an impossible dream that will never see general use. A far better approach is design transformation -- especially, for distributed and concurrent systems. These latter systems are notoriously difficult to reason about, because events occur concurrently and because of the non-determinism of message arrivals. A better rigorous approach than code verification is to 1) design a centralized solution; and 2) use correctness-preserving transformations to build a distributed architecture for the system. The original solution can be verified much more easily than the distributed solution. The transformations need to be verified only once. Observe, also, that all verification in this paradigm is done at the design level. If the design language has constructs that facilitate verification, then a process that seems impossible at code level becomes tractable.

In summary, I object to emphasizing the T&E of implementations of large, complex systems. The focus should be on improving the quality of the design process, so that T&E is less important. This focus is viable, because of the developing theory of design transformations. Design artifacts are much simpler than code and can be reasoned about more easily. Formal transformations offer the hope of correctly mapping designs into detailed designs and implementations.

Of course, implementation-level T&E research should continue, particularly that which emphasizes re-usable modules that can be tested once against an interface specification. As mentioned above, I support empirical investigations of successful practical testing organizations to identify tools that will help automate the testing process. Evaluation research that emphasizes a standardized environment and reused modules and that provides a process for developing customized measures from explicit project goals should be supported.

The greatest research efforts should go into developing a higher-quality design process for distributed and concurrent systems. Design languages compatible with Ada that offer constructs at higher levels of abstraction than PDL must be developed. Clear linkage to Ada programs must be provided, of course, but the design constructs must be sufficiently abstract to allow simplified reasoning. Research on transformational methodologies should be encouraged to develop techniques for constructing distributed architectures from centralized designs.

[Evan84] Evangelist, W.M. 1984. "An Analysis of Control Flow Complexity." In *Proceedings 8th International Computer Software and Applications Conference*. Washington, DC: IEEE Computer Society.

SDS SOFTWARE TESTING AND EVALUATION

H. Dieter Rombach

Dept. of Computer Science & UMIACS
University of Maryland
College Park, MD 20742

1. INTRODUCTION

This paper contains (a) my assessment of the technology gaps in the software evaluation field and suggests possible R&D tasks intended to close these gaps, and (b) a detailed discussion of my past and current work with respect to software evaluation (including experience in the area of large, distributed, real-time systems).

Measurement provides a mechanism for better understanding, predicting and controlling software processes and products. What metrics are of interest depends heavily on the goals and characteristics of a specific project or organization. It is understood that these objectives and characteristics may vary drastically across organizations and even projects within the same organization. Therefore, it is impossible to suggest a specific set of metrics without detailed knowledge regarding the objectives and project characteristics they are supposed to visualize. For the same reason it dangerous to adopt metrics from other environments without careful re-validation. It is necessary to support the proper use of metrics by a comprehensive measurement methodology that assists in (i) defining the analysis and improvement goals of a project, (ii) deriving the appropriate set of metrics that allow the quantification of those project-specific goals, (iii) defining the context for interpreting data and computing metrics, and (iv) allowing for the capturing and reuse of metric-based knowledge in this and future projects of similar type. Such a measurement methodology itself needs to be integrated into an overall process development model that combines construction and analysis in order to benefit the overall project. Only construction oriented activities can improve the quality of the resulting product.

Many of the metric literature seems to contain contradicting results and recommendations. Most of those contradictions are due to the fact that metric results are presented context free. For example, the application of some complexity metric to a small sequential system development effort to predict product size will result in different insights than applying the same complexity metric to a large distributed system development effort intended to predict maintainability. The role of complexity in these two scenarios is entirely different. Nobody would know how to use the metric 'horse power' to predict a car buyer's satisfaction with his/her car without additional information about the car buyer him/herself (e.g., purpose of car, income). In the area of software development (which is obviously more complex than the previous example) much too often metrics are believed to be applicable in such a naive way. I believe that engineering-style software development (planning based on objective information, prediction and control based on facts rather than subjective guessing) requires measurement. The more complex processes are the more important it is to make decisions based on precise information. Developing large, distributed, real-time systems with extreme reliability requirements represents the ultimate combination of challenges.

The major technology gaps in the area of software evaluation are the lack of a sound measurement methodology, the wrong perception that measurement can be viewed as an add-on to software development instead of being an integral part, and as a consequence, the lack of proper automated support via integrated software development environments. The TAME (Tailoring A Measurement Environment) project at the University of Maryland attempts to close these gaps [5].

2. SOFTWARE MEASUREMENT

I am referring to this field as 'software measurement' in order to reflect the fact that there is more to the effective use of measurement than metrics. Measurement is an ideal mechanism for characterizing, evaluating, predicting, controlling, and providing motivation for the various aspects of software development (and maintenance) processes and resulting products. Effective use of measurement requires (i) to understand some basic characteristics of software development and maintenance, (ii) to use a sound measurement methodology, (iii) to integrate this measurement methodology into the respective overall software development process model, (iv) to support all aspects of measurement automatically to the degree possible, and (v) to account for the specific needs and characteristics of the application domain. Sections 2.1. and 2.2. contain my assessment of the current measurement technology gaps and suggested R&D efforts to close them, respectively.

2.1 MAJOR TECHNOLOGY GAPS IN THE MEASUREMENT FIELD

For each of the five topic areas related to measurement I will address some of the related issues in detail, and characterize the existing technology gaps or misconceptions.

2.1.1 Basic Characteristics of Software Development

The basic characteristics of (not only, but in particular) software development that need to be considered for measurement include the relationship between construction and analysis, the relationship between processes and products, and the need for tailoring measurement due to constantly changing project needs:

- **Construction versus Analysis:** It is important to clearly distinguish between the role of construction (e.g., design, code) and analysis oriented (e.g., testing, evaluation) activities. Only improved construction processes will result in higher quality software. Quality cannot be tested or inspected into software. Analytic activities (e.g., testing, evaluation) cannot serve as a substitute for construction oriented activities but will provide better control of the construction oriented activities. We can describe the role of construction versus analysis oriented activities as follows: Construction is oriented towards **preventing** errors; analysis is oriented towards **detecting** and/or **isolating** failures and faults. The **correction** of isolated faults is then again a construction oriented activity. For example, testing (in the sense of validation) is intended to detect failures during execution, debugging is intended to isolate the software faults responsible for a previously detected failure, and code reading (as an off-line reading technique) combines both detection and isolation. Verification (as opposed to validation) is not intended to identify failures or faults but rather their absence. It is obvious that construction and analysis oriented activities need to complement each other. On the one hand, analysis oriented activities should focus on those issues that have not been taken care of by construction properly; on the other hand, construction oriented activities need to make sure to be performed properly based on analyzable processes and to produce analyzable products.

- **Processes versus Products:** Measurement must be taken on both the software processes and the various resulting software products. Both kinds of measurement are equally important. However, as soon as measurement is intended to result in quality improvements (of process or product), measuring the construction process (which is the creator of quality) becomes indispensable. We should distinguish between passive and active uses of measurement. Passive use of measurement means to visualize some product or process property to better understand it. Passive measurement examples include measuring the effort distribution across life-cycle phases so we understand where we spend our money, or measuring the number of lines of code of a product so we have a measure of its size. As soon as measurement results are used for actively impacting an ongoing or future project, we speak of active use of measurement. Active measurement examples include the use of measurement for control, prediction or improvement. Improving (through better methods or tools or better control of their execution) a product requires understanding both the product and its construction process.

- **Needs for Tailoring**: All project environments and products are different in some way. They might differ with respect to overall project goals (e.g., high reliability) that need to be achieved as well as specific characteristics of the project environment (e.g., personnel qualification, development lifecycle model, specific application domain). These differences need to be taken into account when deciding what metrics to use and how to interpret them. These differences among project environments make it impossible to suggest a specific set of metrics without detailed knowledge regarding the specific project goals and characteristics. For the same reason it is dangerous to adopt metrics from other environments without careful re-validation.

The technology gaps are mainly misconceptions regarding the nature of software development. Even if the need for distinguishing clearly between construction and analysis, for measuring both processes and products, and for tailoring measurement (including the set of metrics to be used) to each new project is theoretically accepted, it has not been translated into practice yet. For example, testing techniques are still used in most industrial settings to build quality into the product. The wrong belief exists that testing can fully compensate for the lack of having prevented problems from entering the product early on. Several studies clearly indicate that one pays for this misconception with lower quality and higher cost. Most of the metrics research so far has concentrated on product metrics. One of the reasons is definitely that data collection from products can be automated more easily. The importance of product metrics has been recognized by industry in order to better control their current state-of-the-practice or improve it. The worst misconception exists about the tailoring issue. It is still widely believed (and reflected in several government standards) that a fixed set of metrics will be sufficient for all kinds of projects. This ignores the fact that metrics are intended to visualize the respective processes and products of a project; whenever important project characteristics change the metrics to visualize the changed characteristics might change too. Even worse, people believe that identical metric values should be interpreted identically across environments. For what reason, do we expect that an 'above average fault-rate detected during unit test' should have the same meaning independent of whether (i) unit testing was performed better than normal, (ii) the preceding development activities were performed poorly, (iii) the application problem was new and more complex, or (iv) the personnel was new and inexperienced with respect to the application domain and/or the testing approach. Nevertheless, people are surprised if two different publications (from different environments) present different results derived from using the same metric(s).

2.1.2 Sound Measurement Methodology

A sound measurement methodology needs to support the top-down definition of metrics (starting from overall measurement goals) as well as the bottom-up interpretation of the prescribed data. In addition, data collection and validation need to be addressed. A sound measurement methodology should consist of at least the following five steps: (1) stating measurement goals based upon the needs of the organization, (2) quantify those goals into a set of related questions, and select the set of appropriate metrics to answer each of the posed questions, (3) define a mechanism for collecting and validating data, and perform it, and (4) analyze and interpret the data [5].

- **Stating Measurement Goals**: Measurement goals need to be defined completely. A complete goal definition needs to address its **purpose, perspective and environment** [5]. There are a variety of uses for measurement. The purpose of measurement need to be clearly stated. Measurement purposes include the examination of cost, efficiency, reliability, correctness, maintainability, effectiveness, user friendliness. Measurement needs to be defined from the appropriate perspective. The organization, manager, developer, tester, customer, and user (operator) each view the product and process from different perspectives. Thus they may want to know different things about the project. The conclusion drawn from observing too many failures being missed during system testing, might be manyfold depending what test procedure had been used, whether it had been used right, and what the test personnel's level of experience with this test procedure or application domain was. Consequently, a complete goal definition needs to capture the possibly important environment characteristics.

- **Selecting Appropriate Metrics**: The appropriate set of metrics to support a specific goal should be the final result of defining this goal in an operational way. This definitional process consists of refining the original goal into a set of quantifiable questions. This is the most difficult definitional step because it requires the interpretation of fuzzy terms like quality or productivity within the context of a software development environment. The aim is to satisfy the intuitive notion of a goal as completely and consistently as possible. Although the exact set of questions depends on the specific goal, experience from many applications of measurement suggests that questions need to be asked regarding two major areas: (1) defining the object to be measured (process or product), (2) modeling the quality perspective of interest, and (3) stating feedback hypotheses [5]. As part of the TAME project we have suggested templates for the class of questions that need to be formulated [5]. According to these templates the definition of processes includes questions regarding the **quality of use** (a quantitative characterization of the process and an assessment of how well it is performed) and the **domain of use** (a quantitative characterization of the object to which the process is applied and an analysis of the performer's knowledge concerning this object). The definition of products includes questions regarding the **physical attributes** (a quantitative characterization of the product in terms of size, complexity, etc.), **cost** (a quantitative characterization of the resources expended related to this product in terms of effort, computer time, etc.), **changes and defects** (a quantitative characterization of the errors, faults, failures, and changes related to this product), and **context** (a quantitative characterization of the customer community using this product and their operational profiles). The quality perspective of interest (both for processes and products) includes questions regarding the **major models used** (a quantitative characterization of the quality perspective of interest), and **validity of the model and collected data** (an analysis of the appropriateness of the model and the quality of the collected data). The feedback hypotheses (both for processes and products) includes questions regarding feedback for the purpose of improvement (a quantitative characterization of major problem areas, possible improvements in this or future projects). Such top-down goal definitions eventually define the appropriate set of metrics. In this case we view the derived set of metrics as the operational definition of the original goal. In addition, the result of this process (a goal/questions/metrics graph) allows for traceability between overall measurement goals and supporting metrics. This traceability will enable us to reuse measurement goal definitions as well as the associated data and metrics more effectively across changing environments. The measurement methodology advocates a comprehensive view of measurement. We refer to any formula that allows the quantification of an issue addressed by a question as a metric. For example, 'Lines of Code' is a metric for product size. In some environments, we might be able to predict the 'average maintenance effort' based on this metric. In this context 'Lines of Code' would be considered a 'complexity metric for maintenance' (and consequently interpreted very differently). We distinguish between objective and subjective, as well as direct and indirect, metrics. Objective metrics are absolute, based on a precise computational model. Two people should compute the identical metric values. Typical examples are 'lines of code' or 'number of failures observed during acceptance testing.' Subjective metrics are relative, based on individual estimates or a compromise within a group. Typical examples are 'degree to which a method was used' or 'experience of personnel with respect to the application domain.' Direct metrics allow the project-specific quantification of a quality aspect of interest according to some model. An indirect metric is intended to help predict the expected values of the direct metric. For example, a meaningful direct metric of 'reliability of the maintained product' might be 'number of failures per week of operation.' The indirect metric 'number of failures during system and acceptance testing' might be useful for predicting the value of the directed metric.

- **Collecting and Validating Data:** Multiple mechanisms are needed for data collection and validation. The nature of data to be collected determines the appropriate mechanisms, e.g., manually via forms or interviews, or automatically via analyzers.

- **Analyzing and Interpreting Data & Metrics:** There exists a large body of knowledge regarding the statistical analysis of measurement data. However, the appropriate interpretation of such data or analysis results is only possible in the context of a specific project environment. Interpretation is the crucial part of software measurement. Data simply visualize some aspect of a software process or product. The interpretation of such data really adds to our body of knowledge that might help us to learn what to do better next time.

The technology gaps are that measurement in most environments is still not supported by a sound measurement methodology. Metrics are chosen if the underlying data can be collected easily, even if those metrics do not relate to the implicit measurement goals at all (or at least, it is not clear whether they do). The result of this kind of approach is necessarily disappointment. The suggested measurement methodology [5] has proven to be capable of basing measurement on a sound basis in industrial settings [2, 6, 15, 16, 18, 20, 24, 28].

2.1.3 Integration of Measurement into Software Development

The ultimate purpose of measurement is to create the information and knowledge allowing for the creation of higher-quality products. This improvement objective can only be met if measurement is becoming an integral part of an improvement-oriented software development process model. Such a process model must include support for measurement according to the methodology described in 2.1.2, learning and feedback, as well as tailoring and reuse [5].

- **Improvement Methodology:** In the TAME project, Victor Basili and I use an improvement-oriented software process model that consists of the iteration of the following six steps: (1) characterize the current project environment in order to identify its weaknesses and strengths, (2) define the improvement goals in an operational way (goals –> questions –> metrics), (3) choose the appropriate methods and tools for construction and analysis to fulfill the stated improvement goals, (4) execute the chosen methods and tools, collect the prescribed data and validate it, (5) analyze and interpret the collected data, determine problems, record findings, and make recommendations for improvement, and (6) proceed to step (1) to start the next project, armed with the experience gained from this and prior projects [5]. This so-called TAME process model addresses characterization, planning and execution from both the construction and analysis perspective. The goal/question/metric-based planning mechanism is intended to integrate planning, execution, and interpretation; it also provides the basis for learning, and feedback. Two additional integration issues need to be addressed: (i) how do we specify feedback loops back into heuristic software processes, and (ii) how do we represent the experience gained via measurement so it can be reused in future projects.

- **Learning & Feedback:** Learning and feedback is integrated in such a model in various ways. On-line learning is achieved via application of the goal/question/metric oriented methodology. The results will be stored in some kind of experience base (e.g., database, information base, or knowledge base). Off-line learning is achieved via manipulation of information within the experience base. Such manipulations include the generalization of project-specific information (e.g., by updating base lines), the formalization of information (informal -> schematized -> productized), and the re-structuring of information.

- **Tailoring & Reuse:** All experience gained via measurement would be worthless if we could not use it to improve the ongoing or future projects. As one of the characteristics of software projects suggests, existing experience needs to be tailored before reuse. The degree of tailoring necessary depends on the discrepancies between the environment in which the existing experience had been created and the potential reuse environment. In addition, we have to address typical reuse issues such as identifying reuse candidates, understanding them, and actually using them. At the University of Maryland we have defined a framework for reuse. This framework allows us to characterize any instance of reuse (whether it is product or process related experience, constructive or analytic information) [7].

The technology gaps are that measurement is not viewed as an integral part of software development. The idea is to capture many aspects of software processes and products in quantitative form (via metrics) throughout all projects within an organization, and reuse it within future projects for the purpose of improving quality and productivity. This definition already implies that measurement-based analysis and construction are two highly-interrelated views of all aspects of software development. In most environments measurement is viewed as an activity that can be done in parallel (as an add-on) to construction activities. It is not tied into the planning phase and, therefore, it is hard to (i) make sure that the right data are collected, and (ii) to make any use of those data (feeding them back). As part of the TAME project, Victor Basili and I are using an improvement-oriented software process model which ties together measurement-based analysis and construction through all stages of the life-cycle. Effective use of measurement makes it necessary to worry about learning and feedback (how do we support the accumulation of information and prepare it for future reuse), as well as tailoring and reuse (how do we tailor existing information to the needs of a new project and actually reuse it). Formalizing learning, feedback, reuse, and tailoring is very important. The TAME project is addressing these research issues [5].

2.1.4 Automated Measurement Support

Automated measurement support is essential to deal with the large amounts of information in an economical fashion. We suggest automated support for each of the measurement-related components of the improvement-oriented process model (section 2.1.3).

The technology gaps are that (a) there exist almost no tools supporting measurement, except for pure metric tools. This fact coincides with the misconception about using metrics in isolation which has been criticized in section 2.1.2. Almost no tools for supporting the other aspects of measurement such as planning, evaluation, feedback, learning, etc. exist today. Most of the existing tools such as TAME, AdaMAT, ATVS, SMDC or NOSC are beyond the prototype level.

2.1.5 Domain-dependent Measurement Issues

There exist specific measurement challenges due to the specific characteristics of the application domain (large, distributed, real-time, highly reliable, iterative enhancement development model). This application domain introduces all possible challenges to software development as well as its analysis. Although, the specific areas to be addressed via measurement will depend on the precise goals, the main characteristics of the application domain seem to suggest special emphasis on the measurement of product complexity as early as possible during design (suggested by the attribute 'large'), parallelism, synchronization, and physical distribution (suggested by the attribute 'distributed'), and non-functional requirements issues (suggested by the attribute 'real-time').

The technology gaps are that most measurement work has dealt with systems that do not fall into this application domain. There doesn't exist a large body of knowledge on how to deal (measurement-wise) with 'distributed' and 'real-time' software systems.

2.2 SUGGESTED R&D TASKS IN MEASUREMENT

Before measurement can be used effectively in an environment as challenging as the SDS software development environment, we need to close some of the identified technology gaps. The following two subsections describe some of the necessary near-term and long-term R&D tasks, which are logical consequences of the discussions regarding technology gaps in section 2.1.

2.2.1 Near-term R&D Tasks

The following near-term tasks should be performed over the next 2 or 3 years:

- **Establish a SDS Measurement Organization**: Sound software development requires a measurement and evaluation infra-structure similar to NASA's Software Engineering Laboratory (SEL). Such an organization would be in charge for establishing a organization-specific measurement organization which would initially monitor all projects in this organization. To set up such an environment, the goals of software projects within this environment need to be defined in an operational way, data collection and validation procedures need to be put in place, and automated tools for data collection, validation, storage and retrieval, and data analysis and documentation made available. Although this task might definitely benefit from the experience gained in the SEL, all the tasks have to be repeated in order to make sure that the measurement organization is, indeed, tailored to the specific needs (i.e. goals and environment characteristics).

  Specific subtasks here include

  -> derive the appropriate set of metrics (based on a clear definition of the overall goals)

  -> set up an appropriate measurement database

  -> select or develop measurement tools that automate the collection and validation of metrics to the maximum degree possible

-> conduct case studies or experiments aimed at an understanding of how certain metric values are to be interpreted in the context of the specific goals in the particular environment (in many cases this might mean the revalidation and tailoring of results published in the literature)

- **Select and specify the most promising Life-Cycle Model:** The initial choice of a development process model will be based on common knowledge and specific characteristics of the application domain. The measurement organization will help in validating whether the chosen model is effective, where it has deficiencies, and how it should be improved.

- **Plan for Integrating Construction and Analysis to allow for effective Reuse:** The goal/question/metric mechanism for defining goals in an operational way needs to be formalized. We need to clearly specify software processes from both the analytic and constructive perspective. This would then allow us to formalize feedback paths between both perspectives. Finally, a comprehensive underlying software engineering database needs to be created, which provides for communication among all the construction and analysis activities.

- **Evaluate the Effectiveness of Candidate Methods and Tools:** The selection of effective construction and analysis methods and tools requires knowledge regarding their effectiveness in this particular environment. One of the pieces of information that should be produced through measurement is to develop profiles regarding the effectiveness of methods and tools. Such data allow for improvement of the software development process over time.

2.2.2 Long-term R&D Tasks

The following long-term tasks should be performed over the next 5 to 10 years:

- **Develop a software development environment according to the improvement-oriented process model as suggested in section 2.1.3.:** Such an environment would actually combine support for the construction and analysis aspects of software development and should formalize to a large degree the mechanisms for planning, learning, reuse, feed-back and tailoring. The degree to which these mechanisms can be automated will depend heavily on the experience gained from the previously established measurement organization, but also on the progress in areas such as formal languages for goal specification (allowing for a more traceable specification of improvement goals) and for process specification (allowing for a more formal specification of what we mean by feeding information back into ongoing development processes), database technology (allowing for more natural and tailorable databases), and artificial intelligence (allowing for support of planning as well as interpretation activities).

3. OTHER TESTING & EVALUATION RELATED ISSUES

[There are several important testing and evaluation related issues that must be considered.]

- **Impact of Iterative Enhancement Process Model:** The fact that SDS software will be developed and deployed incrementally suggests some kind of an iterative enhancement software development model. According to such a process model a series of systems get developed and incrementally deployed each satisfying a larger subset of the requirements. The effectiveness of this type of development model depends on the traceability between life-cycle phases. In order to replace one system version by the next one, it needs to be clear where they differ as far as the requirements are concerned, which system components were affected by this change in requirements, and as a result which system components of the previous version have to be replaced and how integration testing needs to be done.

- **Verification versus Validation**: The decision between validation and verification again cannot be made independent of other lifecycle issues. Formal verification is only possible if the object that needs to be verified (e.g., source code) as well as the reference document (e.g., specification) against which it is to be verified are entirely formal. This is another example where construction issues (i.e., the choice of specification approaches) is directly limiting the possible choices for validation and verification. Another limiting factor for verifying the entire software system is the state-of-the-art of verification. It is typically not automatable and therefore very time-consuming. However, it might make sense to verify at least the crucial software parts. Crucial for this type of application is, for example, to achieve the necessary level of reliability. Usually, this is achieved via some kind of fault-tolerance mechanism which is supposed that the system doesn't fail even in the presence of faults. The crucial parts as far as fault tolerance mechanisms are concerned are (i) the ability to detect the malfunction of a component and trigger the fault-tolerance mechanism, and (ii) the protocols based on which malfunctions are detected and fault-tolerance mechanisms are invoked. The first problem requires some kind of good domain and use analysis. However, the second problem should be avoided by verifying these crucial fault-tolerance protocols.

- **System versus Software Aspect**: In the case of real-time systems it doesn't make sense to distinguish between software, hardware or system failures during operation. Even if the software could be proven to be correct, we still need monitoring mechanisms that are capable of identifying malfunctioning system components. Such monitoring mechanisms can either be passive (in that they monitor and evaluate the application dependent actions of the system) or active (in that they create actions themselves in order to test and monitor the system).

- **Use of Evaluation to Derive Development Guidelines**: It should be clear that all the results from testing and evaluation should be translated into guidelines for the development process of the next increment of the iterative development process or future projects.

4. MY PAST AND CURRENT RESEARCH WORK

I include a detailed description of my past, and present research. The unifying theme of all my research work was and is the systematic improvement of software development and maintenance. I am currently doing research in the following five sub-areas: (1) creating a methodology for constructing software for distributed systems, (2) formulating models for systematic analysis (evaluation and improvement) of software processes and products based on measurement, integrating evaluation and improvement into software process models, and supporting such process models automatically, (3) applying the analysis models to a variety of specific analyses tasks of software methods and tools as well as products, (4) creating a comprehensive framework for reuse, and (5) developing a model and language for specifying software processes and products.

1. **Construction Method for Distributed Software:**
   Developing software for distributed systems adds a class of new problems to software engineering (e.g., parallelism, distribution, communication). Different projects around the world attempt to

develop sound software development methodologies for distributed systems. Their approaches can be distinguished based on the software models used, the choice of abstracting from physical distribution during software development or not, and the decision to maximise the available concepts and language features (for better flexibility) or minimize them (for better control). The DISTOS and INCAS projects at the University of Kaiserslautern, West Germany, are aimed at building a comprehensive methodology for developing distributed system software [3,17]. I was part of the team designing the computational and structural INCAS model of system software which became the basis for the implementation language LADY [3], developed a prototype tool supporting the design of distributed software [14], and was in charge of the developing twelve distributed software systems (implemented in LADY) and evaluating them to determine the impact of LADY on quality aspects such as understandability, modifiability, maintainability, and reuse [2]. Currently I have plans to compare the pros and cons of different models for the development of distributed software (e.g., making physical distribution decisions early in the software development process, or dealing only with parallelism during software development and making the physical distribution decisions as late as possible). It is an open question today how to best develop distributed Ada software [22].

2. **Models for Systematic Evaluation and Improvement of Software:**
Measurement has become widely accepted as an important mechanism for capturing and formalizing experience regarding processes and products. Improvement of a project over a previous project is possible if we can reuse experience accumulated during this previous or any other project. It is obvious that any two project may be different with respect to their goals and/or their environment characteristics. Therefore, existing experience needs to be modified or tailored towards the goals and characteristics of the new project before reuse. In order to determine the needs for tailoring effectively we need to know the project context (goals and characteristics) of both the project in which the experience has been accumulated and the project in which it is expected to be reused.

As part of my efforts to evaluate the DISTOS/INCAS approach (see activity (1)), I developed an initial measurement methodology for the purpose of evaluating (project) hypotheses. This model acknowledges the need for tying metrics and the interpretation to precise definitions of analysis goals including the characterization of the project environment. Independently, Victor Basili and his group at the University of Maryland had worked on similar models for years. Since I joined the University of Maryland in 1984, Victor Basili and I work jointly on refining these models, augmenting them with templates guiding the precise specification of goals and identification of supporting metrics, integrating the evaluation model into the overall software development model for the purpose of improvement, and supporting such improvement-oriented process models via automated environments.

The initial analysis model that was used to evaluate the effectiveness of the INCAS approach as far as its impact on a variety of quality aspects of the resulting software was concerned is described in [2]. Basically, the model addresses the problem of choosing the appropriate metrics. It puts the role of metrics into the context of specific evaluation goals and hypotheses. For example, instead of using some unmotivated set of metrics in order to evaluate maintainability, we would have a measurement plan that refines the evaluation objective "evaluate the impact of complexity on maintainability" into a set of metrics grouped into "complexity-related (e.g., number of control paths)", "maintainability-related (e.g., effort per change)", "maintenance-process-related (e.g., the maintenance process followed strictly the following process: xxxxx, effort distributions per process step)", "development-process-related", and "personnel-related (e.g., experience with the maintenance process, experience with the application domain)". The benefits are that we do not only communicate clearly what we mean by "complexity" and "maintainability", we also have a context for interpretation. It is much less likely that we will come up with improper statements generalizing the impact of complexity independent of all the other environment characteristics.

At the University of Maryland, Victor Basili and I jointly head the TAME (Tailoring A Measurement Environment) project which is aimed at further formalizing the initial analysis models, instantiating them into improvement-oriented process models, and building prototypes for automated support of such analysis and process models. In [5] (and [8, 19, 23]) we summarize our joint experience regarding the software development process in general and measurement in specific in the form of lessons learned. These lessons motivated the entire TAME project. In addition to having refined the evaluation and improvement models, we have managed to formalize the goal/question/metric idea (expressing that (i) metrics are derived from overall goals through questions in a top-down process, and (ii) metric values need to be interpreted in the context of these same questions and goals in a bottom-up process) by providing a number of templates for specifying goals and deriving questions and metrics. These templates got developed incrementally as the result of many practical applications of our analysis models. The improvement model is a refinement of the scientific model in the introduction of this report as the basis for my research. It consists of the iteration of characterizing the status quo, planning for improvement, executing the project (including data collection, validation and analysis), learning and feedback. This improvement model has been expanded into an improvement-oriented software development process model by including the construction-oriented activities of planning and execution. A series of (analysis-oriented) environments is being developed supporting such an improvement-oriented process model [5, 10]. Potential users of such environments will be supported in (i) formulating evaluation goals using the template mechanism, (ii) deriving a set of appropriate metrics using the template mechanism, (iii) collecting and validating data, (iv) interpreting collected data, (v) storing and retrieving data and interpretations in the context of goals and project environment characteristics, and (vi) interfacing with planning and/or construction activities of the ongoing or future projects for the purpose of feedback and improvement. We have applied the TAME analysis models to various practical problems, mainly in the areas of testing, maintenance, and quality assurance.

3. **Experimental Analysis of Software Methodologies:**
I have conducted a series of controlled experiments and case studies trying to evaluate various product and process-oriented aspects of software development and maintenance according to the analysis models that resulted from activity (2).

I have conducted a number of additional controlled experiments and case studies aimed at better understanding of the impact of methods and tools which is part of my involvement with the Software Engineering Laboratory (joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation). They range from studies aimed at characterizing the effectiveness of methods and tools used in the SEL environment in a form that allows for tailoring this experience before reuse in a new project [18], to analyzing the relative role of code reading in the software life cycle [24], and identifying the impact of structural characteristics of Ada programs on their maintainability [25]. Especially the study aimed at producing tailorable characterizations of the effectiveness of methods and tools produced valuable input to the reuse-oriented research activity (4). Many of my experimental measurement and evaluation work was and is concerned with maintenance [6, 15, 20] and software quality assurance [1, 4, 27]. Currently, I am setting up a measurement environment intended to monitor maintenance activities at NASA Goddard Space Flight Center. This environment will be an extension of the existing development-oriented SEL. The idea is to use measurement to improve the maintenance process itself as well as feed maintenance lessons learned back into the development process.

4. **Software Reuse:**
Reuse of products and processes will be the key to enabling the software development industry to achieve the dramatic improvement in productivity and quality which is required to satisfy the anticipated growing demands. Although experience shows that certain kinds of reuse can be successful, general success in this area is elusive. Most current reuse efforts concentrate on just product reuse and view reuse as an issue that can be dealt with independent of any particular software development process which is expected to enable reuse. A comprehensive reuse framework which allows broad and extensive reuse could provide the means to achieve the desired order-of-magnitude improvements. Victor Basili and I have developed such a comprehensive reuse framework. In [7] (a refined version of [26]) we describe this framework in terms of its key components: (i) a scheme for systematically characterizing all reuse-related aspects, and (ii) a reuse-oriented software evolution model that addresses how reuse can be supported in the context of software products. The reuse characterization scheme introduces twelve dimensions which are meant to characterize any instance of software reuse. The objective is to take a broad view of reusability in order to capture as many dimensions of the concept as possible. The reuse-oriented software evolution model addresses how learning and reuse can be integrated into software project environments. We suggest five areas of emphasis in order to support the transition from traditional software evolution to reuse-oriented software evolution: (a) mechanisms for recording experience, (b) mechanisms for using (finding, assessing) existing experience, (c) mechanisms for tailoring existing experience, (d) mechanisms for formalizing experience, and (e) mechanisms for integrating learning and reuse mechanisms into an improvement oriented software evolution process model. Each of these five areas is discussed separately. There exist a number of possible approaches towards reuse oriented software evolution, depending on the degree to which learning and reuse are explicitly supported.

   In a project funded by AIRMICS, Victor Basili and I investigate the impact of different degrees of external coupling of components in a reuse library and their impact on reuse effort. External coupling is measured based on a data binding model. In many instances we are capable of applying function-preserving transformations to reduce the external coupling of components. Ultimately, all this research is aimed at deriving development guidelines allowing for the creation of highly reusable components (at least from this coupling point of view) in the first place. The project results are published in detail in [30,31].

5. **Specification of Software Processes and Products:**
In order to analyze a product or process effectively, we need to have a description of this product or process at the desired level of abstraction. In this context, I refer to any description (at any level of detail, abstraction, or formality) as a 'specification'. Obviously, the state-of-the-art in product specification is much more advanced than in process specification. Combined with the importance of better understanding software processes which ultimately enables us to trigger improvement constitutes the urgent needs for means to specify software processes. A growing number of projects is currently addressing this need around the world from many different perspectives (e.g., to support reuse, feedback based upon measurement, generating software engineering environments supporting a specific development approach). I have developed a first prototype language allowing for the specification of process and product types at any level of detail. The idea is to instantiate objects of certain types to control execution of a project, store data accumulated according to project execution, and to provide the process context for reusing experience. Together with Leo Mark from our database group we are developing a software engineering database that allows the automated generation of database schemes from the specifications of the software process that needs to be supported.

In [12,21] I describe a first prototype language that allows the specification of process and product types. The language acknowledges the fact that development processes consist of mechanical and creative tasks by combining algorithmic as well as behavioral language features. It also acknowledges the fact that our degree of understanding of different process aspects is different today by providing a refinement mechanism. We allow the description of each process (e.g., the design process) in terms of it's interface (e.g., input is a document of type 'requirements', output is a document of type 'design', and pre-condition for executing the design process is a condition derived from schedule requirements), and its implementation (e.g., a refinement into sub-activities such as high-level design and low-level design). The key is not to enforce any level of detail that is not understood, but to provide a language that allows the capturing of the existing process knowledge and its future refinement as we learn more. The language has been validated to some degree by applying it to different scenarios (e.g., the maintenance process at NASA/GSFC). The benefits of having a formal language allowing for the capturing of all aspects of software development and maintenance will not only improve our understanding of software development in general, it will also (i) enhance the effectiveness of measurement because feed-back mechanisms back into the process can be formally specified, (ii) change the approach to reuse which in my view cannot be solved without taking the development process (which enables reuse) into account, and (iii) allow the generation of (components of) automated software development environments from specifications of the process they are supposed to support (together with a colleague of mine we have already developed a procedure allowing the generation of database schema from process specifications [9,13]).

5. REFERENCES

1. QUANTITATIVE SOFTWARE-QUALITAETSSICHERUNG: EINE METHODE ZUR DEFINITION UND NUTZUNG GEEIGNETER MASSE (Quantitative Software Quality Assurance: A Method for Defining and Using Proper Quality Measures) (with V. R. Basili), Special Issue on Software Quality Assurance, GI Informatik Spektrum (Journal of the German Computer Society "GI"), vol. 10, no.3, June 1987, pp. 145-158 (invited paper, in German).

2. A CONTROLLED EXPERIMENT ON THE IMPACT OF SOFTWARE STRUCTURE ON MAINTAINABILITY, IEEE Transactions on Software Engineering, Special Issue on Software Maintenance, Vol. SE-12, No.3, March 1987, pp. 344-354.

3. KEY CONCEPTS OF THE INCAS MULTICOMPUTER PROJECT (with J. Nehmer, D. Haban, F. Mattern, and D. Wybranietz), IEEE Transactions on Software Engineering, special issue on 'Software for Local Area Networks', Vol. SE-13, no.8, August 1987, pp. 913-923.

4. SOFTWARE QUALITY ASSURANCE (with V. R. Basili), IEEE Software, special issue on 'Software Quality Assurance', [Guest Editors' Introduction], September 1987, pp. 6-9.

5. THE TAME PROJECT: TOWARDS IMPROVEMENT-ORIENTED SW ENVIRONMENTS (with V. R. Basili), IEEE Transactions on Software Engineering, special issue on 'Software Engineering Environments', vol. SE-14, no.6, June 1988, pp. 758 - 773.

6. IMPROVING SOFTWARE MAINTENANCE THROUGH MEASUREMENT (with Bradford T. Ulery), IEEE Proceedings, special issue on 'Software Maintenance', to be published in April 1989 (invited paper).

7. SOFTWARE REUSE: A FRAMEWORK (with V. R. Basili), submitted to IEEE Computer Magazine, September 1988.

8. TAME: INTEGRATING MEASUREMENT INTO SOFTWARE ENVIRONMENTS (with V. R. Basili), Technical Report TR-1764 (and TAME-TR-1-1987), Department of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

9. A META INFORMATION BASE FOR SOFTWARE ENGINEERING (with Leo Mark), Technical Report TR-1765, Department of Computer Science, University of Maryland, College Park, MD 20742, July 1987.

10. TAME: REQUIREMENTS AND SYSTEM ARCHITECTURE (with V. R. Basili, K. Reed, L. Mark, D. Stotts, and other members of the TAME project), Technical Report (and TAME-TR-3-1988), Department of Computer Science, University of Maryland, College Park, MD 20742, to be published in October 1988.

11. SOFTWARE SPECIFICATION: A FRAMEWORK, Technical Report (Draft), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, to be published as CMU/SEI TR (curriculum module).

12. SOFTWARE PROCESS & PRODUCT SPECIFICATIONS: A BASIS FOR GENERATING CUSTOMIZED SOFTWARE ENGINEERING INFORMATION BASES (with Leo Mark), Technical Report CS-TR-2062/UMIACS-TR-88-51, Department of Computer Science/Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, July 1988. [accepted for HICSS-22, Hawaii, January 1989]

13. GENERATING CUSTOMIZED SOFTWARE ENGINEERING INFORMATION BASES FROM SOFTWARE PROCESS AND PRODUCT SPECIFICATIONS (with Leo Mark), Technical Report CS-TR-2063/UMIACS-TR-88-52, Department of Computer Science/Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, July 1988. [accepted for HICSS-22, Hawaii, January 1989]

14. EXPERIENCE WITH A MIL DESIGN TOOL (with K. Wegener), Proceedings of the Eighth European Conference on Programming Languages and Program Development, Zurich, Switzerland, March 1984.

15. DESIGN METRICS FOR MAINTENANCE, 9th NASA-Workshop on Software Engineering, Goddard Space Flight Center, Greenbelt, MD 20771, November 1984.

16. THE USE AND INTERPRETATION of CHARACTERISTIC METRIC SETS with CHANGE, ERROR, and FAULT DATA (with Richard W. Selby Jr.), NSIA National Joint Conference on Software Quality and Productivity, Williamsburg, March 1985.

17. THE MULTICOMPUTER PROJECT INCAS - OBJECTIVES, CONCEPTS, AND EXPERIENCE, Proceedings of the Pacific Computer Communication Symposium, Seoul, Korea, October 22 - 24, 1985 (invited paper).

18. TAILORING THE SOFTWARE PROCESS TO PROJECT GOALS AND ENVIRONMENT (with V. R. Basili), Proc. of the Ninth International Conference on Software Engineering, Monterey, California, 30 March - 2 April, 1987.

19. TAME: TAILORING AN ADA MEASUREMENT ENVIRONMENT (with V. R. Basili), Fifth National Conference on Ada Technology, Arlington, VA, March 16-19, 1987.

20. A QUANTITATIVE ASSESSMENT OF SOFTWARE MAINTENANCE: AN INDUSTRIAL CASE STUDY (with V. R. Basili), Conference on Software Maintenance, Austin Texas, September 21-24, 1987.

21. A SPECIFICATION LANGUAGE FOR SOFTWARE ENGINEERING PROCESSES AND PRODUCTS, 4th Software Process Workshop, London, UK, May 11-13, 1988.

22. A DEVELOPMENT METHODOLOGY FOR DISTRIBUTED ADA APPLICATIONS (with C. M. Stefanescu, COMPASS, Inc., Wakefield, MA), Fifth Washington Ada Symposium, Tysons Corner, Virginia, June 27-30, 1988.

23. A METHODOLOGY FOR EVALUATING AND IMPROVING LIFE CYCLE SUPPORT BY TECHNIQUES (with V. R. Basili), Eighth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, July 30 - August 2, 1985.

24. THE ROLE OF CODE READING IN THE SOFTWARE LIFE CYCLE (with Victor R. Basili and Richard W. Selby, Jr.), Ninth Minnowbrook Workshop on Software Evaluation, Blue Mountain Lake, New York, August 5-8, 1986.

25. STRUCTURE AND MAINTAINABILITY OF ADA PROGRAMS – Can we measure the Differences – (with Elizabeth E. Katz and Victor R. Basili), Ninth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, August 5-8, 1986.

26. SOFTWARE REUSE: A RESEARCH FRAMEWORK (with V. R. Basili, J. Bailey, and B. G. Joo), Proceedings of the Tenth Minnowbrook Workshop on Software Performance Evaluation, Blue Mountain Lake, New York, July 28-31, 1987.

27. A QUANTITATIVE APPROACH TO SOFTWARE QUALITY ASSURANCE, Proceedings of the Seventh National Symposium on EDP Quality Assurance, Washington, D.C., October 19-22, 1987 (invited paper).

28. WHAT DATA ARE NEEDED FOR MEANINGFUL RELIABILITY ASSESSMENT AND PREDICTION?, Proceedings of the Annual National Joint (NASA, NSIA, et al.) Conference on Software Quality and Productivity, Arlington, Virginia, March 1-3, 1988.

29. THE EVOLUTION OF DESIGN METRICS RESEARCH: A SUBJECTIVE VIEW, Design Metrics Workshop (sponsored by SPS and US Army), Melbourne, Florida, March 14-16, 1988.

30. ADA REUSE METRICS (with Victor R. Basili, John Bailey, Alex Delis and Farhat Farhat), Proceedings of the AIRMICS Workshop on Ada Reusability and Metrics, Atlanta, Georgia, June 15 & 16, 1988.

31. ADA REUSABILITY ANALYSIS AND MEASUREMENT (with V. R. Basili, J. Bailey, and A. Delis), Proceedings of the Sixth Symposium on Empirical Foundations of Information and Software Sciences, Atlanta, GA, October 19-21, 1988.

IDA Testing and Evaluation Workshop: Critical Research Gaps, Research & Development Tasks, and Current Research Summary

Richard W. Selby

Department of Information and Computer Science
University of California
Irvine, California 92717

1. Critical Gaps in Evaluation Technology

Several "critical gaps" in evaluation technology are identified below. After describing a critical gap, an assessment of possible near-term (2–3 years) and long-term (10 years) achievements is given. Specific research and development tasks that would facilitate progress are outlined.

1.1 National Software Engineering Laboratory

A national software engineering laboratory would be the focal point of studies evaluating software engineering methods. Such a laboratory would develop, refine, and recommend available methodologies for data specification, collection, analysis, and interpretation. The laboratory would also house data collected from numerous software projects. The data would span requirements analysis through maintenance and include a full spectrum of metric data (e.g., errors, human effort, changes, process metrics, product metrics). A national software engineering laboratory could be modeled after the Software Engineering Laboratory (SEL) operated by NASA-Goddard, University of Maryland, and Computer Sciences Corporation. The vast collection of software project data would (i) enable studies of factors contributing to project quality and productivity, and (ii) provide a baseline from which predictive models can be customized.

Near-Term (2 to 3 Years) Possible Achievements The initial step is to select the right people to head the lab and provide them with the necessary resources. By the "right people," I mean established leaders in the software evaluation technology field. Their selection will automatically generate visibility for the lab and promote its credibility. Existing efforts, such as SEL and RADC, would be investigated to determine what could be learned and reused. Data from existing collection efforts, plus data from projects currently underway, would provide an initial base of information for analysis.

Long-Term (10 Years) Possible Achievements The lab would be fully underway and would be a central resource for data collection, analysis, and evaluation processes for a wide range of software projects. Data would have been collected on literally hundreds of projects. A wealth of information would be available about effectiveness of particular software engineering methods on different project types, error profiles, operational scenarios, etc.

Specific R&D Tasks Establish a "new" lab in addition to those currently underway or enhance an existing laboratory to be the national software engineering laboratory. Such a lab needs to have visibility and credibility in academic, industrial, and government circles, and hence, should have representatives from each area.

1.2 Series of Evaluative Studies of Software Engineering Methods

Systematic evaluation methodologies need to be applied to assess the effectiveness of proposed software engineering methods, such as those in requirements analysis, testing, and system reconfiguration. A series of studies could be conducted, at the rate of about two major empirical studies per year, to quantify the benefits of using particular methods. The series of studies could be organized by the national software engineering laboratory in Section 1.1.

The studies would incorporate different samples along several dimensions. Some example dimensions are: software development phase; software engineering method; software type (e.g., sequential/concurrent, real-time, command and control); error-fault-failure type; operational profile; and expertise of human developers.

Near-Term (2 to 3 Years) Possible Achievements A base of about 4–6 major empirical studies could be completed. This base of studies would serve as models for later studies, and hence, would reflect experimental methodologies appropriate for different evaluation criteria and different software engineering methods. Software engineering methods that are considered pivotal to SDS success, such as software *testing*, should be evaluated in this initial base of studies.

Long-Term (10 Years) Possible Achievements A broad base of studies across all lifecycle phases and the full spectrum of software engineering methods would be available. The results *from* almost all of the studies would have been confirmed through replication.

Specific R&D Tasks The highest priority "dimensions" need to be identified and samples selected within each dimension. For example, we may decide that determining the relationships among the following dimensions is the highest priority issue: software testing methods, error-fault-failure types, and software types. Then we would select particular testing methods, error-fault-failure types, and software types to be investigated (these are the "samples") and specify an appropriate experimental design. There should be a mix of controlled and field studies. Evaluative studies on different topics could be done in parallel and at different sites. Experts in individual technology areas could provide advice on the organization of studies. Experimentation methodologies and data collection methods could certainly be reused.

1.3 Integrative Frameworks for Using Metrics Synergistically

Too often metrics are used as "stand alone" evaluative measures of software development processes or products. Commonly more than one metric is necessary to characterize or evaluative a process or product. There is no one universal metric; there might be a fixed set of evaluative metrics — but the intuition is that different metrics are required for different purposes. Integrative frameworks for software metrics would assist project developers, managers, end-users, researchers, etc. in the use of multiple metrics in concert. Such frameworks would help users decide which metrics to use, when to use them, how much accuracy can be expected, etc. The frameworks would help identify metrics that capture orthogonal pieces of information. Importantly, the frameworks provide a mechanism for using multiple metrics together synergistically. Section 2.1.1 describes an example integrative framework using decision trees.

Near-Term (2 to 3 Years) Possible Achievements Prototype systems could be available for use by persons other than their developers. It would be useful to have several different (i.e., at least more than one) integrative frameworks for using multiple metrics. Underlying conceptual models would be in various stages of development.

Long-Term (10 Years) Possible Achievements Robust systems would be available for use in software projects or analytical studies.

Specific R&D Tasks Some researchers in evaluation technology are still looking for the "one universal metric." They should broaden their focus to include the use of customized sets of metrics for different quality and productivity criteria on different types of software processes and products. Prototype systems that support alternate integrative frameworks should be developed and evaluated. The evaluation of such systems could use the data collected by the national software engineering laboratory (Section 1.1).

1.4 Support for Metric Specification, Collection, Analysis, and Display in Software Environments

Software environments provide support for a multitude of software development and maintenance activities. Software environments will play an important role in the development of SDS software. Several software environment projects are underway. Metrics are used to characterize, evaluate, predict, control, etc. software projects, and therefore, it is natural for software environments to support software metrics. Software environments need to provide technique(s) for metric specification, collection, analysis, and display. Several features are desirable: transparency of collection details, minimal impact on performance, reuse of collection methods, and minimal interference with human developers. Experts in the graphics and human factors communities could highlight opportunities for best uses of sophisticated graphical display techniques.

Near-Term (2 to 3 Years) Possible Achievements Prototypes of software environments should start being distributed and used by persons other than their developers. (Note that some "environments" from the private sector are available today, but they in general only provide limited sets of capabilities and tools.) The environment prototypes should contain some mechanisms for specification, collection, analysis, and display for some (probably small) set of commonly used metrics. A preliminary version of an integrative framework for using metrics together (see Section 1.3) may also be incorporated in the environment.

Long-Term (10 Years) Possible Achievements Robust software environments would be available for use by projects. The environments would support a wide spectrum of customizable metrics and analysis techniques. The collection methods would be managed by the environment architecture and would impact the environment user and running processes minimally. The metric analysis processes could be managed by a robust integrative framework that uses multiple metrics and statistically analyzes metric data relative to past projects.

Specific R&D Tasks Investigation of alternate methods for metric specification, collection, analysis, and display is required. Examples of several areas needing examination include: using metric taxonomies for specification, using concurrency during collection, using existing statistical packages for analysis, and using graphics for data display. There is also the issue of the interplay between the metric support capabilities and the other environment components such as object managers, type models, user interface management systems, persistent storage managers, etc. Finally, current software environment projects need to incorporate techniques for specification, collection, analysis, and display of metrics.

1.5 Development and Validation of Metrics for Concurrent, Distributed, Real-Time Systems

Of the several product metrics that have been proposed, few have been targeted to concurrent, distributed, real-time systems. Metrics for such systems will be required for SDS software. The development of appropriate metrics meets only part of the need — the metrics need to be validated using data from actual projects to ascertain whether they capture what is intended. Metrics that are available in early development phases, such as metrics based on Ada specifications, would be useful for predictive models.

Near-Term (2 to 3 Years) Possible Achievements An initial set of metrics for characterizing, evaluating, controlling, etc. concurrent, distributed, real-time systems would be defined. The metrics available today, such as size and control flow-based measures, would be extended to their counterparts for concurrent, distributed, real-time systems. Validation studies of the relationships between the metrics and the information they intend to measure would be underway. Development of automated tools to support the collection of the metrics would be underway.

Long-Term (10 Years) Possible Achievements A full range of metrics for concurrent, distributed, real-time systems would be defined and have automated tool support for collection and analysis. Almost all of the metrics would be validated in at least one empirical study. Almost all of the metrics and supporting tools would be available in some software environments.

Specific R&D Tasks Several tasks are needed: motivation of possible metrics by investigation of conceptual models for concurrent, distributed, real-time systems; definition of metrics for concurrent, distributed, real-time systems; development of automated tools for their collection and analysis; and incorporation of the tools into software environments.

2. Summary of My Current Testing and Evaluation Research

2.1 Current Status of My Research and Expected Progress

My research spans several software research areas:

- Software metric decision tree generation [SP] [Sel87a],
- Data interaction metrics [SB88] [Sel88],
- Software metric specification techniques,
- Reuse [Sel] [Sel87b], and
- Evaluation of combinations of fault detection methods [Sel86] [BS87].

2.1.1 Software Metric Decision Tree Generation

Abstract This study investigates one integrative framework for software metrics: decision trees. The decision trees use historical data to identify certain classes of objects, such as software modules that are likely to be fault-prone or costly to develop. The proposed approach uses a recursive algorithm to automatically generate decision trees, whose nodes are multi-valued functions based on software metrics. The decision trees can be used early in a project to identify components (e.g., modules, subsystems) under development that are likely to be error-prone, so that developers can focus their resources accordingly. The decision tree leaf nodes contain a probability (or simply a "yes" or "no") to indicate whether an object is likely to be in a certain class based on historical data. Decision trees are useful structures since they are straightforward to build and interpret. Decision trees use different metrics to classify different types of objects; they do not use just one metric or a fixed set of metrics.

A feasibility study was conducted to evaluate the predictive accuracy of the decision trees. The purpose of the decision trees was to identify classes of objects (software modules) that had high development effort or faults, where "high" was defined to be in the uppermost quartile relative to past data. Sixteen software systems ranging from 3000 to 112,000 source lines were selected for analysis from a NASA production environment. The collection and analysis of 74 metrics, for over 4700 modules, capture a multitude of information about the modules: development effort, faults, changes, design style, and implementation style. A total of 9600 decision trees were automatically generated and evaluated based on several parameters: (i) metric availability; (ii) evaluation function heuristic; (iii) tree termination criteria; (iv) number of projects in the training set; (v) ordinal grouping of metrics; and (vi) dependent variable. Sensitive 2^4x5x15 full-factorial analysis of variance models were employed to assess the performance contributions of the factors and their interactions simultaneously. The analysis focused on the characterization and evaluation of decision tree accuracy, complexity, and composition. The decision trees correctly identified 79.3% of the software modules that had high development effort or faults, on the average across all 9600 trees. The decision trees generated from the best parameter combinations correctly identified 88.4% of the modules on the average.

Current Status Preliminary versions of the decision tree generation and evaluation tools have been implemented. The predictive accuracy of the trees has been analyzed using the NASA data. Analysis of decision tree complexity and metric composition is underway. Refinement of decision tree generation algorithm is underway.

Expected Near-Term Progress Completion of the analysis of decision tree complexity and metric composition using the NASA data is expected. Second version of prototype implementations of decision tree generation and evaluation tools is expected. Evaluation of decision tree tools using data from a large aerospace company will be underway. Investigation of methods for incorporating decision tree tools into software environments will be underway.

Expected Long-Term Progress Completion of evaluation of decision tree tools using data from a large aerospace company is expected. Decision tree generation framework is expected to provide an integration mechanism for applying a full spectrum of metrics in concert.

2.1.2 Data Interaction Metrics

Abstract One central feature of the structure of a software system is the coupling among its components (e.g., subsystems, modules) and the cohesion within them. The purpose of this study is to quantify ratios of coupling and cohesion and use them in the generation of hierarchical system descriptions. The ability of the hierarchical descriptions to localize errors by identifying error-prone system structure is evaluated using actual error data. Measures of data interaction, called data bindings, are used as the basis for calculating software coupling and cohesion. A 135,000 source line system from a production environment has been selected for empirical analysis. Software error data was collected from high-level system design through system test and from some field operation of the system. A set of five tools is applied to calculate the data bindings automatically, and cluster analysis is used to determine a hierarchical description of each of the system's 77 subsystems. An analysis of variance model is used to characterize subsystems and individual routines that had either many/few errors or high/low error correction effort. The empirical results support the effectiveness of the approach for localizing errors. The approach is especially useful during software maintenance since the tools require only the source code for automatic generation of a hierarchical system description.

Current Status Preliminary versions of the data bindings analysis tools have been implemented. Investigation is underway of variations of data interaction metrics, such as alternate coupling and cohesion measures and refinement of coupling/cohesion ratios.

Expected Near-Term Progress Several definitions of data interaction metrics will be available. Research will be underway to make some of the five tools language independent.

Expected Long-Term Progress Conceptual design for minimizing language dependencies in the data bindings tools will be completed. Data bindings metrics will be integrated into decision tree formalism.

2.1.3 Software Metric Specification Techniques

Abstract Research is just getting underway to investigate metric specification techniques. A proposed technique under development is driven by an interactive graphical editor that enables users to specify metrics and analysis processes based on a goal-driven metric taxonomy. The output of the editor is a language whose interpreter manages the data collection.

Current Status The conceptual design for the editor has been defined. Implementation mechanisms are being discussed.

Expected Near-Term Progress Draft of the specification technique methodology, supported by a prototype editor, interpreter, and a small set of metrics, is expected.

Expected Long-Term Progress In-depth discussion of the issues in software metric specification will be articulated. The specification technique methodology will be completed, as will the supporting editor, interpreter, and a commonly used set of metrics.

2.1.4 Software Reuse

Abstract Reusing software may be the catalyst that helps the software community achieve large improvements in software productivity and quality. There are several motivations for desiring software reuse, including gains in productivity by avoiding redevelopment and gains in quality by incorporating components whose reliability has already been established. The purpose of this study is to characterize software reuse empirically by investigating one development environment that actively reuses software. Twenty-five software systems ranging from 3000 to 112,000 source lines have been selected for analysis from a NASA production environment. The amount of software either reused or modified from previous systems averages 32% per project in this environment. Non-parametric statistical models are applied to examine 46 development variables across the 7188 software modules in the systems. The analysis focuses on the characterization of software reuse at the project, module design, and module implementation levels. Four classes of modules are characterized: (a) modules reused without revision, (b) modules reused with slight revision (< 25% changes), (c) modules reused with major revision ($\geq$25 % changes), and (d) newly developed modules. The modules reused without revision tended to be small and well documented with simple interfaces and little input-output processing. Those modules tended to be "terminal nodes" in the projects' module invocation hierarchies and their incorporation required relatively little development effort.

Current Status Preliminary analysis of the software modules reused in each of the four categories is complete. Further analysis of the errors in the reused versus newly developed modules is underway.

Expected Near-Term Progress Completion of analysis of the errors in the reused versus newly developed modules is expected. Use of economic modeling techniques, such as present value analysis and cost accounting, will be investigated as bases for software reuse cost models.

Expected Long-Term Progress Prototype implementations of software reuse cost models will be completed. Analysis and evaluation of software reuse data from several software projects will be underway.

2.1.5 Evaluation of Combinations of Fault Detection Methods

Abstract This study applies an experimentation methodology to compare state-of-the-practice software fault detection strategies: (1) code reading by stepwise abstraction, (2) functional testing using equivalence partitioning and boundary value analysis, and (3) structural testing with 100% statement coverage criteria — and the six pairwise combinations of these techniques. The study compares the strategies in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty two professional programmers applied the techniques to three unit-sized programs in a fractional factorial experimental design. The major results of this study are the following. (1) The six combined testing approaches detected 17.7% more of the programs' faults on the average than did the three individual techniques, which was a 35.5% improvement in fault detection. (2) The highest percentage of the programs' faults were detected when there was a combination of either two code readers or a code reader and a functional tester. However, a pairing of two code readers detected more faults per hour than did a pairing of a code reader and a functional tester. (3) The pairing of two persons of advanced expertise resulted in the highest percentage of faults being detected. (4) The most cost-effective (number of faults detected per hour) testing approach overall was when code reading was applied by a single person. The most cost-effective combined testing approach was when a code reader was paired with either another code reader or a structural tester. (5) Both the percentage of faults detected and the fault detection cost-effectiveness depended on the type of software being tested. (6) Code reading detected more interface faults than did the other individual methods. (7) Functional testing detected more control faults than did the other individual methods. (8) When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.

Current Status Comparison of three individual fault detection techniques is complete. Characterization of the faults found when techniques combined is underway. Analysis of fault detection costs is underway.

Expected Near-Term Progress Characterization of the faults found when techniques combined will be complete. Analysis of fault detection costs will be complete.

Expected Long-Term Progress Paper summarizing studies of the effectiveness of fault detection techniques will be completed. Empirical comparison of fault detection techniques for concurrent software will be underway.

References

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Software Engr.*, SE-13(12):1278–1296, December 1987.

[SB88] Richard W. Selby and Victor R. Basili. *Analyzing Error-Prone System Coupling and Cohesion*. Technical Report, Dept. of Computer Science, University of California, Irvine, 1988.

[Sel] Richard W. Selby. Empirically analyzing software reuse in a production environment. In W. Tracz, editor, *Software Reuse --- Emerging Technologies*, IEEE Computer Society, New York. (in press).

[Sel86] Richard W. Selby. Combining software testing strategies: an empirical evaluation. In *Proceedings of the Workshop on Software Testing*, ACM/SIGSOFT and IEEE-CS Software Engineering Technical Committee, Banff, Canada, July 1986.

[Sel87a] R. W. Selby. Automatically generating software metric decision trees for identifying error-prone and costly modules. In *Proc. of the Twelfth Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, MD, 1987.

[Sel87b] Richard W. Selby. Analyzing software reuse at the project and module design levels. In *Proceedings of the First European Software Engineering Conference*, pages 227–235, Strasbourg, France, September 1987.

[Sel88] Richard W. Selby. Generating hierarchical system descriptions for software error localization. In *Proceedings of the Second Workshop on Software Testing, Verification, and Validation*, Banff, Alberta, Canada, July 1988.

[SP] Richard W. Selby and Adam A. Porter. Learning from examples: generation and evaluation of decision trees for software resource analysis. *IEEE Trans. Software Engr.* (to appear).

Dr. Vincent Shen

MCC
3500 W. Balcones Center Dr.
Austin, TX 78759

CURRENT ACTIVITIES

I am currently a member of the Distributed Computing research group at MCC. Our group develops theories, tools, and methodologies to assist designers of large, complex, distributed systems. Our focus is the "upstream" of the development process; that is, we believe that the design phase offers the greatest opportunity to improve overall software productivity and quality. Our research has led to a new design environment called VERDI, which was recently released to selected development organizations of MCC's Shareholders. VERDI provides a visual editor that enables a designer to specify a design using block diagrams, and an embedded language editor that enables a designer to specify the computation in the block diagrams. It also provides a simulation capability that allows a designer to examine the behavior of the design based on the computation paradigm originally introduced by the Raddle language. The computation paradigm was published in the paper, "Using Raddle to design distributed systems", by Michael Evangelist, Vincent Y. Shen, Ira R. Forman, and Mike Graf in the Proceedings of the 10th International Conference on Software Engineering, April 1988, 102-110. We are currently assessing VERDI's impact on productivity, life cycle time, and quality of the design in one of the Shareholder projects.

I am also the editor of the QualityTime department of the IEEE Software magazine. Articles in the department represent the state-of-the-practice in evaluating software productivity and quality.

GAPS IN SOFTWARE EVALUATION TECHNOLOGY

Mature software evaluation technology will enable software engineers to describe the current state of software parameters, to predict software parameters, to express requirements, and to quantify tradeoffs. While we have had some successes in predicting some software parameters (for example, see "Industrial software metrics top 10 list" by Barry Boehm, IEEE Software, September 1987, 84-85), we generally do not know why the parameters behave the way that has been observed. Productivity or quality models that are based on some complexity metrics are generally NOT significantly better than a linear model of size. Furthermore, no model consistently estimates some parameter to within 25% more than 75% of the time, which fits the intuition for being accurate. Despite these well-known problems, we still have many proprietary tools making claims of accuracy that are seldom substantiated. Software evaluation technology is definitely far from reaching maturity.

APPLYING CURRENT EVALUATION TECHNOLOGY

I do not expect imminent breakthroughs in defining more accurate models for software evaluation. We should train the practitioners in applying the imperfect models to get useful results. One approach is to track the changes of some focused parameters over time. It is often possible to associate the changes with some changes in personnel, environment, methodology, etc. and therefore influence them.

Panel: Reliability Assessment

J. Baldo and K. Gordon, Chair

Panelists: A.F. Ackerman
A. Goel

AT&T SRM RESEARCH REPORT

John D. Musa
A. Frank Ackerman

The purpose of this report is to respond to a request from the Institute for Defense Analyses, Computer and Software Engineering Division, for an opinion on:

- Critical gaps in the field of Software Reliability Measurement (SRM) as they might relate to a sufficient technology base for Strategic Defense Systems; and
- What we might expect to achieve in the near term (2 to 3 years) and the long term (10 years) to close these gaps and how best to facilitate this effort.

We also provide a detailed discussion of current research on SRM being conducted at AT&T Bell Laboratories that includes:

- Critical status and expected progress in the near-term and long-term; and
- How this work might benefit SDS Software T&E.

Critical Gaps

The most critical gap in software reliability measurement with respect to testing and evaluation of SDS software is the need to develop an algorithm that will adjust software reliability measurements for the use of an operational profile during test that is different than the one that will be encountered during field use. The operational profile is basically the set of all the different kinds of "runs" the system will make and the probabilities of occurrence of each [1]. The use of a different operational profile during test will be necessary for SDS because of the need to maximize testing efficiency.

A second gap relates to failure severity classification. Presently we can measure overall software reliability in the operational environment with excellent accuracy. Often, however, overall reliability is not as important as the reliability of certain failure classes. Generally we divide all the possible ways in which a system can fail to meet its requirements into different severity classes of failure. Usually critical failures are so rare that we can gauge reliability with respect to these critical failures only by measuring reliability across all failures and then proportioning that overall measurement among different failure classes. For this procedure to be valid, the class proportions must remain relatively constant. Our data in this area is limited; much more is needed to determine if the proportions are indeed constant. If not, an appropriate model must be developed to deal with this.

There is another critical gap not directly relevant to testing and evaluation but one whose solution might be of considerable benefit to SDS from a political support and funding viewpoint. This is the capability to predict software reliability in the system engineering phase or any phase prior to system test. Since there have been many questions concerning whether SDS could provide an acceptable level of reliability in its software, technology that could help answer this problem would be vital. Such prediction is dependent on being able to predict the parameters of software reliability models prior to execution of the software.

Currently we have a method for predicting the parameters of the basic execution time model based on estimates of the size of the source code, the source code fault density, the size of the object code, the speed of the processor, and the values of the fault exposure ratio and fault reduction factor [1]. However, with current data this method is only accurate within one order of magnitude, we need:

1. A study (using data) of numerous projects to determine if (as it currently appears) the fault exposure ratio and fault reduction factor are nearly constant or if not, what factors influence them.
2. Additional techniques for predicting the parameters of other models, especially the logarithmic Poisson model.
3. The ability to measure and use information about requirements, specifications, design descriptions, and code complexity, and information from software inspections and unit test to further refine our parameter prediction as we proceed through the development cycle.

It should be noted that there are two other areas where methods exist that could be improved and refined. Although these are not critical areas, research would clearly be cost effective in improving SDS test and evaluation:

1. improved software reliability model parameter estimation;
2. study of software reliability model resource usage parameters [1] and factors that influence them, based on project data.

Closing the Gaps

None of the problems cited above appears to be especially intractable. The research tools are mathematical analysis, simulation, and statistical analysis of actual data. The key to solving these problems is the close interplay of these three elements. This approach requires a team of researchers with strong mathematical analysis and statistical skills in conjunction with broad experience in software development. Above all it requires class collaborations between theoreticians and real-life software projects.

The major time constraint is the time it takes to complete the software development life cycle for a major project. With sufficient resources progress could be made on all the problems listed above in 2 to 3 years. We could expect that within 10 years we could develop a body of theory and technique that would allow software reliability measurement to be routinely and accurately applied to all but the most esoteric of software development situations.

In summary, the proposed research tasks are:

1. develop an algorithm for adjustment of software reliability measurements made with an operational profile in test that is different from the one expected in operation,
2. study stability of failure severity class proportions,
3. study software reliability parameter prediction prior to program execution,
4. study improved software reliability model parameter estimation, and

5. study software reliability model resource usage parameters.

Software Reliability Research at AT&T Bell Laboratories

Current software reliability research & development at AT&T Bell Laboratories falls into five areas:

1. developing an algorithm for adjusting software reliability measurements for differences in operational profile,
2. studying stability of failure severity class proportions,
3. studying software reliability parameter prediction prior to program execution,
4. studying improved software reliability model parameter estimation, and
5. solving specified application problems as they are encountered in use on actual projects (several are actively using software reliability measurement).

Note that the first four topics correspond directly to the research tasks recommended for SDS. This should not be surprising, because research needed for application of the technology is not really project-dependent.

Topic 1 is being actively pursued, and there is a good chance of a solution within one year. Work on Topics 2 and 3 has just started and is proceeding on a low level. They both depend on extensive data collection. Topic 4 is being pursued at a moderate level.

References

[1] J. D. Musa, A. Jannins, K. Okumoto, Software Reliability: Measurement, Prediction, Application, *McGraw-Hill*, 1987.

Dr. Amrit Goel

Syracuse University
ECE Dept.
111 Link Hall
Syracuse, NY 13244

SOME COMMENTS ON SOFTWARE RELIABILITY ASSESSMENT

The current methodology for evaluating software reliability is based on a very restricted premise, viz, the future error occurrence phenomenon is a stochastic extrapolation of the recent past. This approach is too simplistic and is not likely to be very useful for ultra-high reliability systems, such as SDS. A more realistic approach should explicitly incorporate software product and process metrics as well as prior subjective or objective informations about the components of the system.

Towards the goal, long term theoretical and experimental R and D efforts will be needed for reliability assessment which should address issues such as the identification of the appropriate metrics and the development of a theoretical framework.

The importance of workload in assessing reliability has not been addressed adequately in the field. It seems to me that the concept of representative operational usage needs to be explicitly quantified and incorporated in the reliability assessment activity. This can be accomplished via short term R & D tasks. Efforts also are needed to address time based workload.

Within the framework of current approach to reliability assessment, additional research is needed in the areas of model validation, parametric estimation, and sensitivity analysis.

Another relevant issue not currently addressed explicitly is that of program execution time as a measure of reliability. Further R and D on this aspect is also recommended.

SUMMARY OF CURRENT WORK

My work is primarily oriented towards explicitly incorporating workload in software reliability assessment and is currently funded by NASA (Langley). We have developed a relationship between average workload and failure time using Wald's equation. We have also been modeling the failure time process where the workload is subject to stochastic changes using the ARIMA class of models.

Another research activity is concerned with the execution time distribution and reliability modeling of fault tolerant programs that use NVP and RBS.

Author Index

Distribution List for IDA Document M-513

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Lt Col Charles Lille SDIO Phase 1 Program Office The Pentagon Washington, DC 20301-7100	1
LTC William Reichert SDIO T&E Room 1E149, The Pentagon Washington, DC 20301-7100	1
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Captain Robert Roncace SDIO T&E Room 1E149, The Pentagon Washington, DC 20301-7100	1
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
CSED Review Panel	
Dr. Dan Alpert, Director Program in Science, Technology & Society University of Illinois Room 201 912-1/2 West Illinois Street Urbana, Illinois 61801	1
Dr. Barry W. Boehm DARPA 1400 Wilson Blvd. Arlington, VA 22209-2308	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1
Dr. John M. Palms, President Georgia State University University Plaza Atlanta, GA 30303	1
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1
Mr. Keith Uncapher University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1
IDA	
General W.Y. Smith, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Mr. Bill R. Brykczynski, CSED	10
Ms. Anne Douville, CSED	1
Dr. John F. Kramer, CSED	1
Mr. Terry Mayfield, CSED	1
Ms. Katydean Price, CSED	2

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Richard Wexelblat, CSED	1
Ms. Christine Youngblut, CSED	2
IDA Control & Distribution Vault	3